Norwegian University of
Science and Technology

Permeability Upscaling Using the DUNE-Framework

Building Open-Source Software for the Oil Industry

Arne Rekdal

Master of Science in Physics and Mathematics
Submission date: June 2009
Supervisor: Helge Holden, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Problem Description

This master thesis will consist of developing a C++ code for upscaling of permeability. Permeability upscaling is a recurring exercise in the oil industry, and should be regarded as an infrastructure tool in reservoir property modelling. However the availability of such tools is unsatisfactory in that open-source alternatives are scarce. This thesis will contribute such a tool to everyone by building on the GPL numerics framework DUNE and by releasing the produced code from this thesis under the GPL license.

The code should be able to handle regular grids, with anisotropic permeability as input at each grid-cell. The upscaling is based on Darcy's law, and will produce a permeability tensor representing the upscaled permeability property.

Assignment given: 15. January 2009
Supervisor: Helge Holden, MATH

Preface

This master thesis is written during the spring 2009 as part of the Master of Science study program in Applied Mathematics at NTNU. My supervisor at Department of Mathematical Sciences at NTNU has been Professor Helge Holden.

I would like to thank my supervisor and Department of Mathematical Sciences at NTNU for making it possible to participate on the DUNE workshop in Heidelberg March 23-27, 2009. This workshop was very useful, and I got the possibility to discuss some of the details in my implementation with the developers of DUNE. I would especially like to thank Peter Bastian, Christian Engwer and Markus Blatt for answering my questions put on DUNE's mailinglist.

At last I would like to thank my supervisor Ph.D. Håvard Berland at StatoilHydro for great help and feedback. I would also like to thank Ph.D. Alf Birger Rustad and Ph.D. Vegard Kippe at StatoilHydro for interesting discussions.

Trondheim, June 12, 2009

Arne Rekdal

Abstract

In this thesis an open-source software for permeability upscaling is developed. The software is based on DUNE, an open-source C++ framework for finding numerical solutions of partial differential equations (PDEs). It provides functionality used in finite elements, finite volumes and finite differences methods.

Permeability is a measure of the ability of a material to transmit fluids, and determines the flow characteristics in reservoir models. Permeability upscaling is a technique to include fine-scale variations of the permeability field in a coarse-scale reservoir model. The upscaling technique used in this thesis involves solving an elliptic partial differential equation. This is solved with mixed and hybrid finite element methods. The mixed method transforms the original second order PDE into a system of two linear equations. The great advantage with these methods compared with standard finite element methods is continuity of the variable of interest in the upscaling problem. The hybrid method was introduced for being able to solve larger problems. The resulting system of equations from the hybrid method can be transformed into a symmetric positive definite system, which again can be solved with efficient iterative methods.

Efficiency of the implementation is important, and as for most implementations of PDE solvers, the computational time is dominated by solving a system of linear equations. In this implementation it is used an algebraic multigrid (AMG) preconditioner provided with DUNE. This is known to be efficient on system arising from elliptic PDEs. The efficiency of the AMG preconditioner is compared with other alternatives, and is superior to the others. On the largest problem investigated, the AMG based solver is almost three times faster than the next best alternative.

The performance of the implementation based on DUNE is also compared with an existing implementation by Sintef. Sintef's implementation is based on a mimetic finite difference method, but on the grid type investigated in this thesis, the methods are equivalent. Sintef's implementation uses the proprietary SAMG solver developed by Fraunhofer SCAI to solve the linear system of equations. SAMG is 58% faster than DUNE's solver on a test case consisting of 322 200 unknowns. The scalability of SAMG seem to be better than DUNE's AMG as the problem size increases. However, a great advantage with DUNE's solver is 50% lower memory usage measured on a problem consisting of approx. $3 \cdot 10^6$ unknowns. Another advantage is the licensing of the software. Both DUNE and the upscaling software developed in this thesis is GPL licensed which means that anyone is free to improve or adjust the software.

Contents

1 Introduction **1**

2 What is DUNE? **3**
2.1 License . . . 4
2.2 Modules . . . 4

3 Permeability Upscaling **7**
3.1 Calculating Upscaled Permeability . . . 8
3.2 Boundary Conditions . . . 8

4 Mixed Finite Element Method **11**
4.1 Strong Form . . . 11
4.2 Weak Form . . . 12
4.3 Discretization . . . 12
4.4 Comments . . . 16

5 Mixed Hybrid Finite Element Method **17**
5.1 Weak Form . . . 17
5.2 Discretization . . . 19
5.3 Implementation Details . . . 21
5.4 Comments . . . 24

6 Mixed Hybrid FEM with Cuboid Shaped Elements **25**
6.1 Discretization . . . 25
6.2 Implementation Details . . . 26
6.3 Installation and Usage . . . 29

7 Algebraic Multigrid (AMG) **31**
7.1 What is AMG? . . . 31
7.2 Two-Level Grid Cycle . . . 32

8 Numerical Results **35**
8.1 Verification of the Implementations . . . 35
8.2 Upscaled Permeability of a Core Sample . . . 42
8.3 Efficiency . . . 43
8.4 Comments . . . 46

9 Conclusion **49**
9.1 Suggestions for Further Work . . . 50

Bibliography **51**

A Appendix **53**
A.1 Two Layered Eclipse Model 53
A.2 Code Listing 54

Chapter 1

Introduction

The motivation for this thesis is to develop an open-source program for permeability upscaling in the oil industry. Permeability upscaling is a technique used in the process of creating reservoir models. The resulting program will be licensed under the GNU General Public License (GPL) [1]. The intention with the GPL License is to guarantee your freedom to share and change free software, and make sure the software is free for all its users. With free it is meant freedom and not price.

The use of GPL license can be a nice way to open for closer contact between academia and industry. To be open and share knowledge is important in the academia, while in the industry, secrecy is often used in order to have a lead to the competitors. With GPL licensed software it is easier for the academia to involve in projects with the industry since the software will be available for everyone.

The method used for permeability upscaling involves solving a partial differential equation (PDE). It is chosen to use a finite element method (FEM) for solving the PDE numerically. StatoilHydro already have a software for calculating upscaled permeability developed by Sintef. Sintef's implementation is used as a measure when the efficiency of the implementation is discussed.

The time spent solving a PDE numerically is often dominated by solving a system of linear equations. The implementation developed by Sintef uses the proprietary linear solver, SAMG, developed by Fraunhofer SCAI. The code developed in this thesis is based on DUNE, a GPL-licensed `C++` framework for solving partial differential equations. The DUNE framework also includes linear algebra solvers, and the efficiency of these will be an important topic in this thesis.

The possibility to create a competitive open-source alternative for permeability upscaling based on DUNE will also be discussed in this thesis.

Chapter 2

What is DUNE?

DUNE is a C++ framework for finding numerical solutions of partial differential equations with grid based methods. DUNE provides basis functionality as traversing the grid and implementations of data structures for sparse matrices and vectors. The DUNE framework is module based, and each module can have dependencies to other modules. The newest module available is the `dune-pdelab` module. This is a module for discretizing PDEs, and is designed to simplify implementations of PDE solvers. It is only required to describe the PDE and the choice of discrete approximation space. Most technical details are hidden for the user. The module was presented for the first time at the DUNE Workshop in Heidelberg in March 2009.

DUNE is based on the following main principles [2]:

- Separation of data structures and algorithms.
- Implementations use generic programming techniques.

The first principle makes sure it is not necessary for the algorithm to know how the data is represented as long as the data structure implements the required interface for the algorithm. For instance, the interface for a conjugate gradient solver needs to know how to perform a matrix-vector product. How the data is represented is not of importance for the solver. This results in less code since the algorithms are implemented for a generic data structure, and not one implementation for each data representation. The choice of data structure is often problem dependent, and is an important factor when it comes to efficiency of the implementation.

The second principle is solved by having the data structure as a parameter for the algorithms. At compilation the compiler performs optimization for the chosen data structure, and the resulting code will be nearly as efficient as it would be with a specialized code. This is called *template programming*, a technique used in large parts of the standard C++ library.

One should avoid optimizing the code for a specific platform. This approach makes the code difficult to maintain since it is a rapid development of hardware and software. Instead it is better to leave the optimizations to the compiler. It is more rational to use resources on optimizing the compilers, instead of optimizing a program for several platforms. In this thesis it is observed that the optimization of template functions are much better in the later versions of the GPL licensed g++ compiler than the prior. Ideally it should only be required to recompile the program without changing the source code to achieve a program with high performance on a new platform. The GNU C++ compiler has potential to be efficient on most platform, since it is possible for everybody to contribute to the project.

2.1 License

The DUNE library and headers are licensed under version 2 of the GNU General Public License (GPL), with a special exception for linking and compiling against DUNE, the so-called “runtime exception”:

> As a special exception, you may use the DUNE library without restriction. Specifically, if other files instantiate templates or use macros or inline functions from one or more of the DUNE source files, or you compile one or more of the DUNE source files and link them with other files to produce an executable, this does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not however invalidate any other reasons why the executable file might be covered by the GNU General Public License.

This license is identical to the license of `libstdc++` which is the `C++` standard library. This makes the DUNE framework attractive to use since the executable does not necessarily inherit the GPL license, and the commitments it involves. According to the GPL license, a public program utilizing parts of GPL licensed software are obligated to distribute the source code with the program, a feature essential to GPL's success in for example the Linux kernel. This condition makes use of GPL licensed software in proprietary software difficult.

There is also another license from GNU called LGPL (Lesser GPL), which is intended to use for software libraries. Programs linked to libraries licensed with LGPL are not imposed any obligations. DUNE consists mostly of template and inline functions, and these are expanded in the code using these functions. The source code from the library is inserted into the source code of the program at compilation. Inclusion into proprietary code would thus not be possible with a LGPL license, even though the intensions are similar. This is why the GNU GPL with runtime exception license is chosen for DUNE.

The choice of license for DUNE is important in order to reach a great market. The choice of license makes it possible for proprietary software companies to use DUNE as a platform, or as a component in a numerical PDE solver. Commercial users can be an important factor for further development of DUNE. Since DUNE is GPL licensed, any improvements or modifications of DUNE source code need to be GPL licensed. The easiest way to improve the DUNE code without any obligations from the GPL license is thus by contributing to the DUNE project.

2.2 Modules

DUNE is module based and contains three core modules:

- `dune-common` contains basic functionality which is used by the other DUNE-modules. This module contains functionality as fixed size vectors, timer and exceptions. A typical user of DUNE will probably not use many of the functions from this module directly, but the other modules are built on functionality of this module.

- `dune-grid` is a large module which includes some easy grid implementations. A DUNE grid can be multi-element-type, multi-level and parallel. The module has methods for graphical output of the grid and data associated with the grid. It

supports file output to IBM data explorer and the VTK file format (parallel XML format for unstructured grid) [3].

- `dune-istl` - *Iterative Solver Template Library.* Sparse matrix and vector classes and solvers. Krylov based solvers are also included.

The `dune-grid` and `dune-istl` modules are only dependent of `dune-common`. In addition there are external modules available:

- `dune-disc` contains shape functions, discretization schemes and other functionality used in finite element methods.
- `dune-fem` is an alternative implementation of finite element methods.
- `dune-pdelab` provides discretization schemes for finite element methods.
- `DuMu`^{x} is (going to be) a multi-scale multi-physics toolbox for the simulation of flow and transport processes in porous media.

In this thesis the three core modules are used. The grids used are unstructured grids of tetrahedra and structured grids of cuboids implemented in the `dune-grid` module. `dune-istl` is used for data representation and for solving the resulting system of equations.

Chapter 3

Permeability Upscaling

Permeability is a measure of the ability of a material to transmit fluids [4]. In reservoir models this parameter is of importance, since the permeability determines the flow characteristics of the fluid. The permeability can be measured directly through Darcy's law,

$$Q = -\frac{KA}{\mu}\frac{P_b - P_a}{L}, \tag{3.1}$$

where $Q = vA$ is the discharge, μ is the dynamic viscosity of the fluid. P_a and P_b are the pressures at position a and b respectively, K is the permeability. The negative sign is required since the flow is pressure driven, and the flow goes from high pressure to low pressure. See Figure 3.1. The permeability can be measured in laboratories using various techniques.

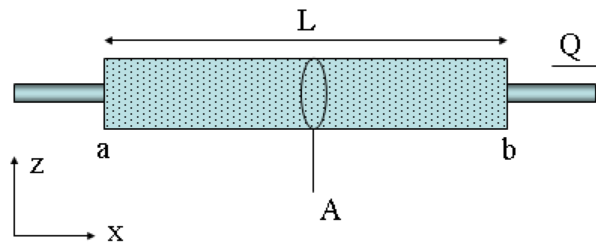


Figure 3.1: Diagram for the variables used in Darcy's law[1].

In a reservoir, the permeability field, $K(x)$, has heterogeneous variations at many length scales that must be represented somehow in a computer model of the reservoir. The challenge is to include information about the fine-scale variations in a coarse model since it is not possible with today's computer resources to work with a fine-scale model. This is the motivation for the upscaling technique.

In the process of creating a reservoir simulation model, upscaling is done in several steps. The first step can be from pore scale ($\sim 10^{-3}$ m) to core scale ($\sim 10^{-2}$ m). The geological models are often of high resolution, and it may be required to create coarser simulation models with upscaled parameters.

[1] Figure by Peter Kapitola published under the Creative Commons Share Alike license [5].

The model consists of grid cells where the petrophysical properties within each cell is constant. Typical size of a grid cell used in simulation a model is 10–50 m in the horizontal directions and 0.1–10 m in the vertical direction. The geological models are often of much higher resolution. Because of limitation in computational resources, it is impossible to use the geological models as simulation models.

3.1 Calculating Upscaled Permeability

An upscaled permeability tensor can be found by solving the following partial differential equation with different boundary conditions,

$$\begin{aligned} \nabla \cdot (-K(x)\nabla p) &= 0 \quad \text{in } \Omega \subseteq \mathbb{R}^3. \\ \frac{\partial p}{\partial n} &= f_N \quad \text{on } \Gamma_N \\ p &= g_D \quad \text{on } \Gamma_D. \end{aligned} \tag{3.2}$$

$K(x)$ denotes the permeability field, p the pressure and $v = -K\nabla p$ the velocity. The equation above is Darcy's law combined with the incompressible fluid condition ($\nabla \cdot v = 0$).

To calculate an upscaled permeability tensor, three sets of boundary conditions are imposed, one for each coordinate direction. The boundary condition for direction η are defined such that the net pressure drop in the η direction is 1. If the numerical velocity solutions, v^η, of (3.2) is found, it is possible to calculate the upscaled permeability tensor,

$$\widetilde{K} = \begin{bmatrix} k_{xx} & k_{xy} & k_{xz} \\ k_{yx} & k_{yy} & k_{yz} \\ k_{zx} & k_{zy} & k_{zz} \end{bmatrix}, \quad \text{where } k_{\xi\eta} = Q_\xi^\eta \Delta_\eta, \quad \xi, \eta = x, y, z. \tag{3.3}$$

Q_ξ^η is the net flow velocity in the ξ-direction when a pressure drop is imposed in the η-direction. Δ_η is the average distance between opposite faces in the η-direction. For example, Q_z^x is found by

$$Q_z^x = \frac{1}{2|\partial\Omega^{\text{top}}|} \int_{\partial\Omega^{\text{top}}} v^x \cdot n \, dS - \frac{1}{2|\partial\Omega^{\text{bottom}}|} \int_{\partial\Omega^{\text{bottom}}} v^x \cdot n \, dS \tag{3.4}$$

n is the outward pointing unit normal on $\partial\Omega$, and $\partial\Omega^{\text{top}}$ and $\partial\Omega^{\text{bottom}}$ are the top and bottom faces of Ω.

3.2 Boundary Conditions

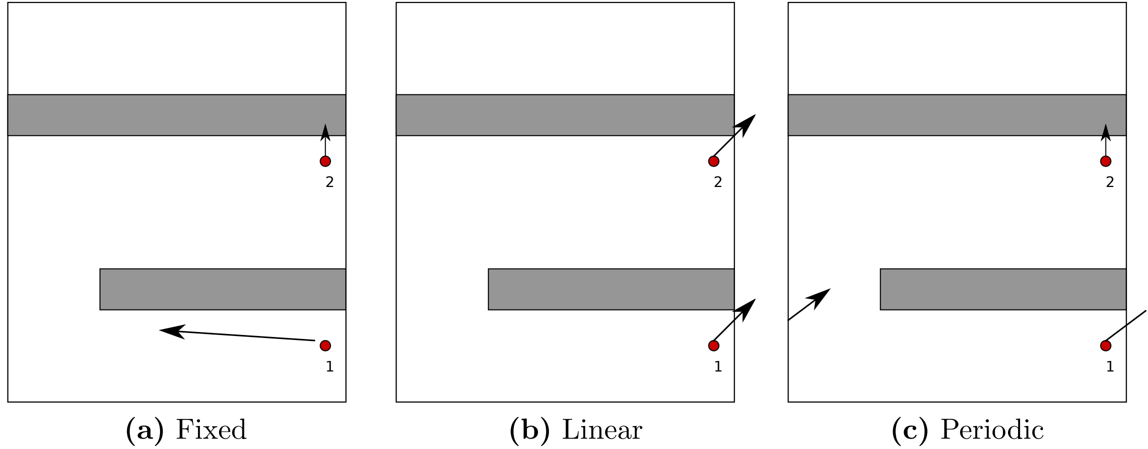
In the upscaling problem the following boundary conditions are often used:

Fixed: The pressure is 1 at one side, and 0 on the opposite side. On all other sides, the no-flow condition applies ($v \cdot n = 0$). This gives a diagonal permeability tensor.

Linear: The pressure is 1 at one side and decays linearly to the opposite side on all boundaries. This gives a full, and in general a non-symmetric permeability tensor.

Periodic: Periodic boundary conditions connects boundaries at opposite sides, but it is required to impose a unit pressure drop in the η direction to generate flow. These boundary conditions try to model flow in an infinite domain. The resulting upscaled permeability tensor is full and symmetric.

$$\begin{aligned} (v^\eta \cdot n)|_{\partial\Omega^{\xi,1}} &= -(v^\eta \cdot n)|_{\partial\Omega^{\xi,2}} \\ p^\eta|_{\partial\Omega^{\xi,1}} &= p^\eta|_{\partial\Omega^{\xi,2}} + \delta_{\xi,\eta} \end{aligned}$$



(a) Fixed **(b)** Linear **(c)** Periodic

Figure 3.2: Overview of the three boundary conditions. The arrows indicate the velocities at two chosen points for each boundary condition. Grey areas represent sections with low permeability compared with the white area. Pressure drop is from bottom to top.

In Figure 3.2 the three different boundary conditions are shown. The fixed condition makes it impossible to flow through the left and right boundaries. With the linear condition, it is possible to flow through the left and right boundaries. In point 1, this is acceptable, since the barrier is not completely tight. In point 2 the flow also goes through the right boundary. This is not realistic, since the barrier is almost tight. This would lead to an unrealistic high upscaled permeability.

The periodic boundary condition is a way to model an infinite large model. In point 1, the flow goes through the right boundary and reappears on the left boundary through the opening in the barrier.

The upscaling problem is equivalent to calculating the effective resistance with Ohm's law. In Figure 3.3 a model with two homogeneous horizontal layers is shown, where the permeability in the top layer is 4 mD and in the bottom layer is 0.01 mD. The upscaled permeability in the horizontal directions equals the volume weighted arithmetic average of the permeability, while the upscaled permeability in the vertical direction equals the volume weighted harmonic average. This is analogue to the effective resistance when several resistors are connected in serial and parallel respectively.

$$\widetilde{K}(1,1) = \widetilde{K}(2,2) = \frac{1}{V}\sum_{i=1}^{2} V_i K_i = \frac{1}{2}(0.01 + 4) = 2.005.$$

$$\widetilde{K}(3,3) = \frac{V}{\sum_{i=1}^{2} \frac{V_i}{K_i}} = \frac{2}{\frac{1}{0.01} + \frac{1}{4}} = 0.01995.$$

In the horizontal direction the flow will mainly go through the top layer. The layers will not interfere with each other, and the effective horizontal permeability is an average of the two layer's permeability. In the vertical direction, it is a barrier as in Figure 3.2a. The flow will be very slow through the bottom layer. This is why the effective vertical permeability is much closer to the permeability in the bottom layer than the top layer.

The model shown in Figure 3.3 is used in all implementations to verify the correctness, since the analytical solution of the upscaled permeability tensor is known.

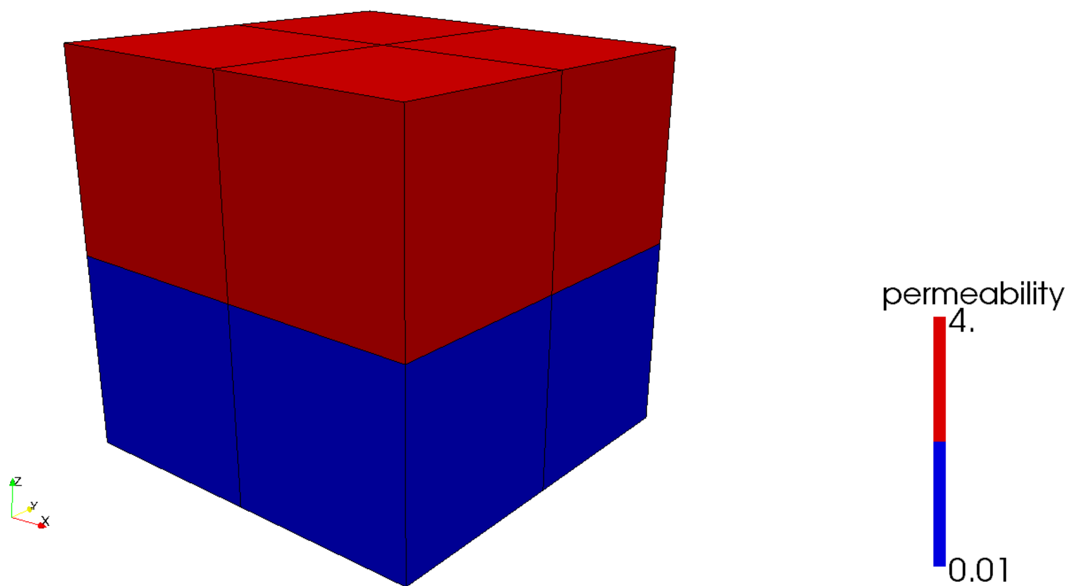


Figure 3.3: Model with two homogeneous layers.

Chapter 4

Mixed Finite Element Method

The mathematical problem discussed in this thesis comes from Darcy's law and local mass conservation, and describes a stationary, single phase flow in a porous media. The problem is stated in (3.2). A mixed finite element method is used for solving the equation as this results in a continuos velocity field, while with ordinary finite element methods this is not necessarily true.

In the mixed finite element method, the equation described in the previous chapter is split into a system of two linear PDEs. The approximation space for the velocity can be chosen to be continuous on the triangulation. The method is described in detail in [6].

4.1 Strong Form

The single phase flow in a porous media can be formulated as [7],

$$
\begin{aligned}
\nabla \cdot (-K(x)\nabla p) &= 0 \quad \text{in } \Omega \subseteq \mathbb{R}^3 \\
\frac{\partial p}{\partial n} &= f_N \quad \text{on } \Gamma_N \\
p &= g_D \quad \text{on } \Gamma_D.
\end{aligned}
\tag{4.1}
$$

This is a generalized Poisson equation with Dirichlet and Neumann boundary conditions. For the permeablity upscaling problem, the variable of interest is not the pressure, p, but the velocity field, v. From Darcy's law, $v = -K\nabla p$, (4.1) can be written as a system of two linear PDEs.

$$
\begin{aligned}
v &= -K(x)\nabla p \quad \text{in } \Omega \subseteq \mathbb{R}^3 \\
\nabla \cdot v &= 0 \quad \text{in } \Omega \\
v \cdot n &= g_N \quad \text{on } \Gamma_N \\
p &= g_D \quad \text{on } \Gamma_D.
\end{aligned}
\tag{4.2}
$$

This is the basis for the mixed method, and reflects the two unknown functions p and v. The solution spaces for p and v are obviously different since p is a scalar function, and v is a vector function.

4.2 Weak Form

The following function spaces need to be defined in order to derive the weak formulation of (4.2):

$$
\begin{aligned}
L^2(\Omega) &= \left\{ u \,\middle|\, \int_\Omega |u|^2 \, d\Omega < \infty \right\}, \\
H^{\text{div}}(\Omega) &= \left\{ u \in \left(L^2(\Omega)\right)^3 \,\middle|\, \nabla \cdot u \in L^2(\Omega) \right\}, \\
H_N^{\text{div}}(\Omega) &= \left\{ u \in H^{\text{div}}(\Omega) \,\middle|\, u \cdot n = g_N \text{ on } \Gamma_N \right\}, \\
H_0^{\text{div}}(\Omega) &= \left\{ u \in H^{\text{div}}(\Omega) \,\middle|\, u \cdot n = 0 \text{ on } \Gamma_N \right\}.
\end{aligned}
\tag{4.3}
$$

The space $H^{\text{div}}(\Omega)$ is larger than $H^1(\Omega)^3$, because the vector functions do not require three continuous components to be in $H^{\text{div}}(\Omega)$, but only continuity of the normal component.

The derivation of the weak formulation of (4.2) is like the derivation of the weak formulation of (4.1), i.e. multiply the equations by a test function, $u \in H_0^{\text{div}}(\Omega)$, and integrate over the domain Ω. The first equation in (4.2) becomes

$$
\begin{aligned}
\int_\Omega u K^{-1} v \, d\Omega &= -\int_\Omega u \nabla p \, d\Omega \\
&= \int_\Omega p \nabla \cdot u \, d\Omega - \int_{\partial\Omega} p u \cdot n \, dS, \quad \forall u \in H_0^{\text{div}}(\Omega).
\end{aligned}
\tag{4.4}
$$

In the same way, the second equation in (4.2) becomes

$$
\int_\Omega q \nabla \cdot v \, d\Omega = 0, \quad \forall q \in L^2(\Omega). \tag{4.5}
$$

By defining

$$
\begin{aligned}
b(u, v) &= \int_\Omega u K^{-1} v \, d\Omega, \\
c(v, p) &= \int_\Omega p \nabla \cdot v \, d\Omega, \\
d(v, \pi) &= -\int_{\partial\Omega} \pi v \cdot n \, dS,
\end{aligned}
\tag{4.6}
$$

the weak form of (4.2) can be formulated as:

Find $p \in L^2(\Omega)$ and $v \in H_N^{\text{div}}(\Omega)$ that satisfies

$$
\begin{aligned}
b(u, v) - c(u, p) &= d(u, g_D), \quad \forall u \in H_0^{\text{div}}(\Omega) \\
c(v, q) &= 0, \quad \forall q \in L^2(\Omega).
\end{aligned}
\tag{4.7}
$$

4.3 Discretization

The domain Ω is covered by a regular triangulation consisting of N non-overlapping tetrahedra,

$$
\mathcal{T} = \bigcup_{i=1}^{N} T_i.
$$

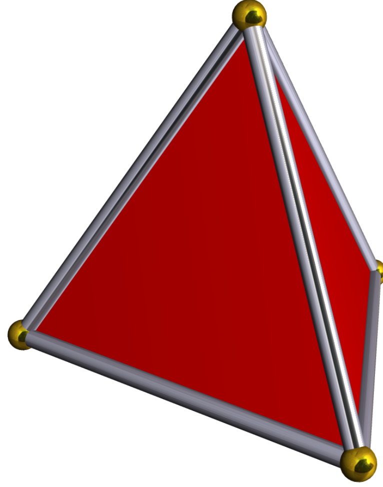


Figure 4.1: A tetrahedron, the 3D simplex[2].

The set of faces in Ω is called $\mathcal{F}_\Omega$, and the set of interior faces in Ω is called $\mathcal{F}_\Omega^{\text{int}}$. The velocity, v, is discretized by using the lowest-order Raviart-Thomas space [8],

$$RT_0(\mathcal{T}) = \Big\{ q \in (L^2(T))^3 \;\Big|\; \forall T \in \mathcal{T}\ \exists\, a \in \mathbb{R}^3, \exists\, b \in \mathbb{R},\ \forall x \in T,\ q(x) = a + bx \\ \text{and } \forall F \in \mathcal{F}_\Omega^{\text{int}},\ [q]_F \cdot n_F = 0 \Big\}, \tag{4.8}$$

where $[q]_F = \big(q|_{T_+} - q|_{T_-}\big)\,|_F$ denotes the jump of q across the face F shared by the elements T_+ and T_-. In words, $RT_0(\mathcal{T})$ is a set of 3 dimensional vector functions with continuos normal component on all interior faces. The components of a vector function is elementwise linear functions.

Continuity of the normal components on the interior faces reflects the conformity, $RT_0(\mathcal{T}) \subset H^{\text{div}}(\Omega)$. To make sure that this continuity is fulfilled, it is built into the shape functions, ψ_F for $F \in \mathcal{F}_\Omega$.

The pressure is discretized by letting the pressure be constant within each element, i.e. $p|_T \in P_0(T)$ for all elements $T \in \mathcal{T}$.

The discrete subspaces read,

$$\begin{aligned} V^N &= \Big\{ u \in RT_0(\mathcal{T}) \;\Big|\; u \cdot n = g_N \text{ on } \Gamma_N \Big\}, \\ V^0 &= \Big\{ u \in RT_0(\mathcal{T}) \;\Big|\; u \cdot n = 0 \text{ on } \Gamma_N \Big\}, \\ P &= \Big\{ q \in L^2(\Omega) \;\Big|\; q|_T \in P_0(T),\ \forall T \in \mathcal{T} \Big\}. \end{aligned}$$

[2]Acknowledgement to Robert Webb's Great Stella software as the creator of this figure, http://www.software3d.com/Stella.html

The discrete problem becomes:

Find $p \in P$ *and* $v \in V^N$ *satisfying*

$$\begin{aligned} b(u,v) - c(u,p) &= d(u,g_D), \quad \forall u \in V^0 \\ c(v,q) &= 0, \quad \forall q \in P. \end{aligned} \tag{4.9}$$

4.3.1 Construction of Face-Basis Functions

It is necessary to create basis-functions for $RT_0(\mathcal{T})$ such that the set of basis functions span the whole vector function space. Each basis function has support in at most two elements, since a face only can be shared by two elements in a conforming grid. From the definition of $RT_0(\mathcal{T})$ it is required that the normal component of $u \in RT_0(\mathcal{T})$ is continuous on each interelement face. The local definition of the basis function in an element must be such that the resulting global basis function has a continuous normal component on the face.

Let F_1, F_2, F_3 and F_4 be the faces of the tetrahedron T opposite to its vertices x_1, x_2, x_3 and x_4 respectively. Let n_{F_α} denote the unit normal vector of F_α chosen with a global fixed orientation while n_α denotes the outer unit normal vector of T along F_α. The basis function for F_α can be defined as

$$\psi_{F_\alpha}(x) = \begin{cases} \frac{\sigma_\alpha |F_\alpha|}{3|T|}(x - x_\alpha) & \text{for } x \in T, \\ 0 & \text{elsewhere.} \end{cases} \tag{4.10}$$

where $\sigma_\alpha = n_\alpha \cdot n_{F_\alpha}$. σ_α is $+1$ if n_{F_α} points outward of the element T, and -1 otherwise. $|T|$ denotes the volume of T, and $|F_\alpha|$ denotes the area of F_α.

Let the face F be shared by the elements T_+ and T_-, such that $\sigma_F = +1$ in T_+ and -1 in T_-. If $x_\pm$ denotes the vertex opposite to face F in $T_\pm$, it can be shown that the global basis function associated with face F becomes [9].

$$\psi_F(x) = \begin{cases} \frac{\pm|F|}{3|T_\pm|}(x - x_\pm) & \text{for } x \in T_\pm, \\ 0 & \text{elsewhere.} \end{cases} \tag{4.11}$$

In [9] it is also shown that the following holds:

1. $\psi_F \cdot n_F = \begin{cases} 0 & \text{along } (\cup\mathcal{F}_\Omega) \setminus F, \\ 1 & \text{along } F. \end{cases}$
2. $\psi_F \in H^{\text{div}}(\Omega)$
3. $\{\psi_F \big| F \in \mathcal{F}_\Omega\}$ is a basis of $RT_0(\mathcal{T})$
4. $\text{div } \psi_F = \begin{cases} \pm\frac{|F|}{|T_\pm|} & \text{for } x \in T_\pm, \\ 0 & \text{elsewhere.} \end{cases}$

Since property 3 holds, it is possible to write any $u \in RT_0(\mathcal{T})$ as

$$u(x) = \sum_{j=1}^{|\mathcal{F}_\Omega|} u_j \psi_j(x).$$

Property 1 above makes it easy to have an interpretation of the coefficient u_F related with the basis function ψ_F. u_F is the flux density through face F.

4.3.2 System of Discrete Equations

Since any member of P is elementwise piecewise constant, $p \in P$ can be written as

$$p(x) = \sum_{i=1}^{N} p_i \chi_i(x), \qquad \text{where } \chi_i(x) = \begin{cases} 1 & \text{for } x \in T_i, \\ 0 & \text{elsewhere.} \end{cases} \tag{4.12}$$

Without loss of generality, let the Neumann condition be $v \cdot n = 0$ on Γ_N as for the fixed boundary conditions in the permeability upscaling problem. The set,

$$\Phi = \mathcal{F}_\Omega \setminus \left\{ F \in \mathcal{F}_\Omega \,\middle|\, F \subset \Gamma_N \right\},$$

is the set of faces not part of the Neumann boundary. The approximate solution for the velocity, $v \in V^N$, can be written as

$$v(x) = \sum_{j=1}^{|\Phi|} v_j \psi_j(x).$$

By inserting these two expressions into the discrete formulation and choosing u and q systematically the result is the following linear system of equations,

$$\sum_{j=1}^{|\Phi|} \underbrace{\int_\Omega \psi_i K^{-1} \psi_j \, d\Omega}_{B_{ij}} \; v_j - \sum_{k=1}^{|\mathcal{T}|} \underbrace{\int_{T_k} \nabla \cdot \psi_i \, d\Omega}_{C_{ki}} \; p_k = \underbrace{- \int_{\partial\Omega} g_D \psi_i \cdot n \, dS}_{(b_d)_i}, \quad \text{for } i = 1, 2, ..., |\Phi|$$

$$\sum_{j=1}^{|\Phi|} \underbrace{\int_{T_k} \nabla \cdot \psi_j \, d\Omega}_{C_{kj}} \; v_j = 0, \quad \text{for } k = 1, 2, ..., |\mathcal{T}|,$$

or in matrix-form notation,

$$\begin{aligned} Bv - C^T p &= b_d \\ Cv &= 0. \end{aligned}$$

The total system of equations to be solved for is

$$\begin{bmatrix} B & -C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} v \\ p \end{bmatrix} = \begin{bmatrix} b_d \\ 0 \end{bmatrix}. \tag{4.13}$$

This system is not positive definite, and thus excludes the possibility to use many of the iterative methods available for solving system of equations.

4.3.3 Local Matrix Blocks

In standard finite element methods it is common to assemble the global matrix system, or the matrix blocks B and C, by iterating over the elements and adding up the element's contribution to the global blocks. This approach is also used here.

The local matrix blocks for an element T are defined as,

$$(B_T)_{jk} = \int_T \psi_j \, K^{-1} \psi_k \, d\Omega, \quad \text{for } j,k = 1,2,3,4. \tag{4.14}$$

$$C_T = \left[\int_T \nabla \cdot \psi_1 \, d\Omega, \int_T \nabla \cdot \psi_2 \, d\Omega, \int_T \nabla \cdot \psi_3 \, d\Omega, \int_T \nabla \cdot \psi_4 \, d\Omega \right]. \tag{4.15}$$

From the choice of basis functions and property 4 in section 4.3.1, it is easy to calculate C_T. B_T can be calculated by transforming the integral into the barycentric coordinate system.

Let λ_1, λ_2, λ_3 and λ_4 denote the barycentric coordinates in the tetrahedron T. ψ_j becomes

$$\psi_j(x) = \frac{\sigma_j |F_j|}{3|T|} \left(\lambda_1(x)(x_1 - x_j) + \lambda_2(x)(x_2 - x_j) + \lambda_3(x)(x_3 - x_j) + \lambda_4(x)(x_4 - x_j) \right).$$

After this transformation it is easier to calculate B_T

$$\begin{aligned}(B_T)_{jk} &= \int_T \psi_j \, K^{-1} \psi_k \, d\Omega = \sigma_j \sigma_k \frac{|F_j||F_k|}{9|T|^2} \sum_{l=1}^{4} \sum_{m=1}^{4} \int_T \lambda_l \cdot (x_l - x_j) K^{-1} \lambda_m \cdot (x_m - x_k) \, d\Omega \\ &= \frac{\sigma_j |F_j| \sigma_k |F_k|}{9|T|^2} \sum_{l=1}^{4} \sum_{m=1}^{4} (x_l - x_j) \cdot (x_m - x_k) \int_T \lambda_l K^{-1} \lambda_m \, d\Omega\end{aligned}$$

By utilizing the transformation, $d\Omega = |T| d\lambda$, and assuming K^{-1} is constant within each element, $K^{-1}(T)$,

$$\int_T \lambda_l K^{-1} \lambda_m \, d\Omega = K^{-1}(T) \frac{|T|}{20} \left(1 + \delta_{lm} \right).$$

The expression for B_T becomes

$$(B_T)_{jk} = K^{-1}(T) \frac{\sigma_j |F_j| \sigma_k |F_k|}{180|T|} \left(\left(\sum_{l=1}^{4} (x_l - x_j) \right) \cdot \left(\sum_{m=1}^{4} (x_m - x_k) \right) + \sum_{l=1}^{4} (x_l - x_j) \cdot (x_l - x_k) \right).$$

4.4 Comments

In the implementation of this method, (4.13) is solved with the direct solver, `SuperLU`. The high complexity of the solver, $\mathcal{O}(N^3)$, for N unknowns implies that the implementation scales poorly with the problem size. This method is only suitable for relative small problem sizes. One advantage with this method is that it is relative easy to implement.

In order to solve the system of equations with iterative methods, Lagrange multipliers are introduced. This modification will be treated in the next chapter.

Chapter 5

Mixed Hybrid Finite Element Method

The problem stated in the previous chapter was solved with a direct LU-solver. This approach is known to be inefficient as the problem size increases. For sparse systems arising from PDE, iterative methods like the conjugate gradient method are attractive. Some iterative methods, including CG, often require the system matrix to be positive definite. The system matrix in (4.13) is not positive definite [10] and restricts the number of linear algebra solvers which can be used.

The requirement of a continuos normal component of the velocity on the interelement faces, is relaxed in this method. Instead, the continuity is weakly imposed by introducing Lagrange multipliers. The continuity across the faces is imposed as a set of constraints. This is advantageous for the resulting system of equations. The motivation for introducing the Lagrange multiplier technique or the hybrid method is to generate a positive definite system of equations which can be solved with the conjugate gradient method.

5.1 Weak Form

The word mixed in the name of the method reflects the two unknowns in the problem, which are v and p. Hybrid corresponds to the constraints associated with continuity at the faces between neighbouring finite elements.

The basis for the derivation of the mixed hybrid FEM is the weak form of the mixed problem derived in section 4.2. The domain Ω is still covered by the triangulation, $\mathcal{T}$, consisting of N non-overlapping tetrahedra. Recall that the members of the previous declared vector function space, V^0, has normal component which are continuous across the interelement boundaries, and $u \cdot n$ vanishes on Γ_N. P is the set of elementwise constant functions. Recall the weak form of the mixed problem:

Find $(v,p) \in V^N \times P$ *such that*

$$\begin{aligned} b(u,v) - c(u,p) &= -\int_{\partial\Omega} g_D u \cdot n \, dS, \quad \forall u \in V^0 \\ c(v,q) &= 0, \quad \forall q \in P. \end{aligned} \tag{5.1}$$

To derive the hybrid formulation it is necessary to define

$$\begin{aligned}\Pi &= \left\{\mu \in L^2(\partial\mathcal{T}) \,\middle|\, \mu \in P_0(F),\ \forall F \in \mathcal{F}_\Omega\right\},\\ \Pi_D &= \left\{\mu \in \Pi \,\middle|\, \mu = g_D \text{ on } \Gamma_D\right\},\\ \Pi_0 &= \left\{\mu \in \Pi \,\middle|\, \mu = 0 \text{ on } \Gamma_D\right\},\end{aligned} \tag{5.2}$$

i.e. the set of piecewise constant functions on all faces of the triangulation, and such functions fulfilling the inhomogeneous and homogeneous Dirichlet boundary conditions respectively.

It is also necessary to define

$$\begin{aligned}RT_0(T) &= \left\{u \in (P_1(T))^3 \,\middle|\, u(x) = a + bx,\ \text{for } x \in T,\ \text{and } a \in \mathbb{R}^3, b \in \mathbb{R}\right\},\\ V^* &= \left\{u \in (L^2(\Omega))^3 \,\middle|\, u|_T \in RT_0(T),\ \forall T \in \mathcal{T}\right\},\\ V^0 &= V^* \cap H_0^{\text{div}}(\Omega).\end{aligned} \tag{5.3}$$

The difference between members of V^* and V^0 is that V^* is larger since it also include functions with discontinuous normal components across interior faces.

It is convenient to redefine for $u \in V^*$ and $\mu \in \Pi$,

$$d(u,\mu) = \sum_{T\in\mathcal{T}} \int_{\partial T} \mu u \cdot n \, dS. \tag{5.4}$$

In [10] it is shown that if $u \in V^*$, then

$$(d(u,\mu) = 0,\ \forall \mu \in \Pi_0) \quad \Leftrightarrow \quad u \in V^0. \tag{5.5}$$

Based on the result above, it is also shown that,

$$b(u,v) - c(u,p) + \underbrace{\int_{\partial\Omega} g_D u \cdot n \, dS}_{d(u,g_D)} = -d(u,\pi_0),\ \forall u \in V^*, \tag{5.6}$$

where $\pi_0 \in \Pi_0$ and is unique. Notice the change of which test-functions the equation above is valid for. Let π_D be defined as

$$\pi_D \in \Pi_D, \qquad \text{where} \qquad \pi_D \equiv \pi_0 \text{ on } \mathcal{F}_\Omega \setminus \Gamma_D. \tag{5.7}$$

(5.5) and (5.6) yields:

Find $(v,\ p,\ \pi) \in V^* \times P \times \Pi_0$ *such that*

$$\begin{aligned}b(u,v) - c(u,p) + d(u,\pi) &= -\,d(u,g_D) \quad \forall u \in V^*,\\ c(v,q) &= 0 \quad \forall q \in P,\\ d(v,\mu) &= 0 \quad \forall \mu \in \Pi_0,\end{aligned} \tag{5.8}$$

The unknown Lagrange multiplier vector, π, corresponds to the pressure at the element's faces. The last equation above comes from (5.5), and states the continuity constraint on the interelement faces. It is important to notice that the solution, $(v,\ p)$, of this problem is identical to the solution of (4.7). For more details regarding the derivation of the hybrid formulation, see [10].

5.2 Discretization

The domain Ω is divided into a disjoint partition of N non-overlapping tetrahedra as for the previous method. The members of V^* has four degrees of freedom per element.

5.2.1 Constructing the Matrix Blocks

Since the Lagrange multiplicators are introduced, the continuity across the faces is not necessary to be built into the shapefunctions as for the face-oriented basis in section 4.2. Each basis function also has local support, i.e. it is non-zero in only one element. The 4 basis functions for a tetrahedron is chosen to be [9]

$$\begin{aligned} \psi_1(x) &= [1,\ 0,\ 0]^T \\ \psi_2(x) &= [0,\ 1,\ 0]^T \\ \psi_3(x) &= [0,\ 0,\ 1]^T \\ \psi_4(x) &= (x - x_c), \end{aligned} \tag{5.9}$$

where x_c denotes the centroid in the element.

The element matrices for a tetrahedron $T \in \mathcal{T}$, B_T and C_T are given by

$$\begin{aligned} (B_T)_{jk} &= \int_T \psi_j K^{-1} \psi_k \, d\Omega, \quad \text{for } j,k = 1,...,4 \\ (C_T)_j &= \int_T \nabla \cdot \psi_j \, d\Omega, \quad \text{for } j = 1,...,4. \end{aligned} \tag{5.10}$$

The choice of basis functions yields

$$\begin{aligned} B_T &= K^{-1} \cdot \operatorname{diag}\left(|T|,\ |T|,\ |T|,\ \int_T |x - x_c|^2 \, d\Omega \right) \\ C_T &= [\, 0,\ 0,\ 0, 3|T| \,]\,. \end{aligned} \tag{5.11}$$

The global matrix B will be of dimension $4N$ where N is the number of elements in the domain Ω. C will be of dimension $N \times 4N$. The global matrices are assembled like

$$B = \begin{bmatrix} B_1 & 0 & \dots & 0 \\ 0 & B_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & B_N \end{bmatrix} \quad \text{and} \quad C = \begin{bmatrix} C_1 & 0 & \dots & 0 \\ 0 & C_2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & C_N \end{bmatrix}. \tag{5.12}$$

The $(\Pi_T)_i$ vector for the face F_i in the element T is calculated by

$$(\Pi_T)_{ij} = \int_{F_i} \psi_j \cdot n_i \, dS, \quad \text{for } i,j = 1,...,4. \tag{5.13}$$

The global Π matrix is assembled with the help of the map

$$\eta_T : \{1,2,3,4\} \to I_{\mathcal{F}_\Omega},$$

i.e. the local face index of the tetrahedron T is mapped to a global face index in $I_{\mathcal{F}_\Omega}$. This means that the global matrix Π can be assembled by

$$\Pi \binom{\eta_T(i)}{4(\text{index}(T) - 1) + \{1,2,3,4\}} = (\Pi_T)_i. \tag{5.14}$$

By similar derivation as in section 4.3.2 the system of equations for (5.8) now reads,

$$\begin{bmatrix} B & -C^T & \Pi^T \\ C & 0 & 0 \\ \Pi & 0 & 0 \end{bmatrix} \begin{bmatrix} v \\ p \\ \pi \end{bmatrix} = \begin{bmatrix} b_D \\ 0 \\ b_N \end{bmatrix}. \tag{5.15}$$

This system is still not an improvement of (4.13). The system size is actually larger than the original. However the choice of V^* makes B block diagonal and each block is of size 4. In addition, with this choice of basis functions, B will actually be diagonal. The importance of this property can be seen in the next section.

5.2.2 Schur-Complement Reduction

The system of equations (5.15) is indefinite as for the previous method. By performing a Schur-complement reduction with respect to B, the result is the positive-definite system [7],

$$\begin{bmatrix} D & -F^T \\ F & -\Pi B^{-1}\Pi^T \end{bmatrix} \begin{bmatrix} p \\ \pi \end{bmatrix} = \begin{bmatrix} -CB^{-1}b_D \\ b_N - \Pi B^{-1} g_D \end{bmatrix}, \tag{5.16}$$

where $D = CB^{-1}C^T$ and $F = \Pi B^{-1}C^T$. Since B is diagonal, B^{-1} will also be diagonal and trivial to compute. The Schur-complement reduction could also be applied the system in (4.13), but finding the inverse of B would be an expensive operation. D is also a diagonal matrix since $C_{ij} \neq 0$ only for $j = 4i$.

$$\begin{aligned} D_{ij} &= \sum_{k=1}^{4N} C_{ik}(1/B_{kk})C_{kj}^T = \sum_{k=1}^{4N} \frac{C_{ik}C_{jk}}{B_{kk}} = \frac{C_{i,4i}C_{j,4i}}{B_{4i,4i}} \\ &= \delta_{ij}\frac{(C_{i,4i})^2}{B_{4i,4i}}. \end{aligned} \tag{5.17}$$

By performing yet another Schur-complement reduction for (5.16) with respect to D, this results in the following symmetric, positive-definite system for π

$$S\pi = r, \quad \text{where} \quad \begin{aligned} S &= \Pi B^{-1}\Pi^T - FD^{-1}F^T \\ r &= (\Pi - FD^{-1}C)B^{-1}b_D - b_N. \end{aligned} \tag{5.18}$$

The advantage with this system is that it can be solved with many iterative methods as the conjugate gradient method which is used in the implementation in this thesis. Another advantage is that it is smaller than the original system since the unknowns are only the pressure at the faces. Once π is found it is relative cheap to find the other unknowns, p and v. p is found by performing a back-substitution in (5.16).

$$p = D^{-1}(F^T\pi - CB^{-1}b_D).$$

This is relative cheap since it only requires inverting a diagonal matrix and calculating three sparse matrix-vector products. When p is found it is possible to find v, which is the variable of interest in the upscaling problem. This is found by solving for v in (5.15),

$$v = B^{-1}(b_D + C^T p - \Pi^T\pi).$$

Finding the velocity vector has the same complexity as finding p.

5.3 Implementation Details

This section is an overview over how the code is organized, and shows how some of the details are solved with use of DUNE. Currently there are no implementation of Raviart-Thomas basis functions and mixed finite element functionality within DUNE, so the whole assembly process for the Raviart-Thomas elements had to be implemented. There is a module under development called `dune-pdelab`, which makes it possible to create a solver for a generic problem, just by specifying the partial differential equation with boundary conditions and the function spaces to use in the discretization. The reason for why `dune-pdelab` was not used was that it was not available before the end of March, and then the implementations was almost finished. In addition, `dune-pdelab` currently only deal with Dirichlet boundary condtitions [11].

5.3.1 Code Structure

The existing DUNE implementation for the Laplace equation with piecewise linear elements is used as a template for how the code for this solver should be organized [12]. The two most important classes are the local assembler class and the assembler class. A local assembler class for the Raviart-Thomas basis is implemented. An object of this type calculates B_T, C_T and Π_T for an element T. The assembly of the global matrix system is done with the assembler class, which uses the local assembler object to fill the global system as mentioned in the previous section. The assembler class also takes a `Problem` object as a parameter where the problem is specified. The `Problem` class has methods for specifying the boundary conditions and the permeability field K. See Figure 5.1.

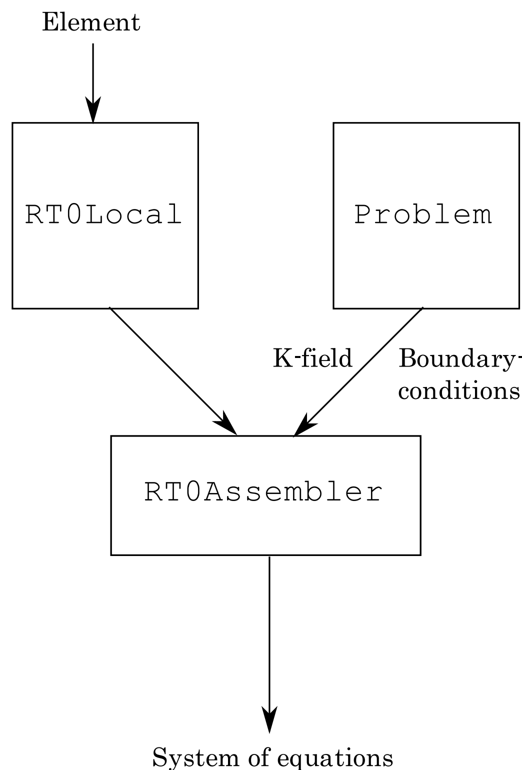


Figure 5.1: The structure of the code. The element is already implemented in DUNE.

The assembler class has also three other methods, schurReduction, solvePressure and solveVelocity. The schurReduction-method performs the transformation of the system of equations, and generates the system in (5.18). This system can be solved with any compatible linear solver. Once the Lagrange multiplier vector, π, is found, the other variables can be found by calling solvePressure and solveVelocity, respectively.

5.3.2 BCRSMatrix

The matrix system arising from solving partial differential equations are often very sparse, i.e. few elements in the matrix are non-zero. If there are n unknowns in the matrix system, the memory requirement for storing the system as a standard matrix will be $\mathcal{O}(n^2)$. By not explicitly saving all the zero elements, it is possible to achieve a lower memory requirement. This is known as a sparse matrix format.

In the implementation of the solver, the BCRSMatrix in DUNE is used to represent sparse matrices. BCRS stands for Block Compressed Row Storage. This is a sparse block matrix with row storage model, i.e. the blocks in a row are stored sequentially in the memory. In the solvers used in this thesis, the "blocks" are scalars implemented as a FieldMatrix<double,1,1>, i.e. 1×1 matrices where the single matrix element is represented as a double.

The usual way of implementing a compressed row storage matrix is by storing three arrays, values, col_index and row_ptr. Let say we want to store the matrix $A \in \mathbb{R}^{6\times 6}$ in CRS format, and

$$A = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}.$$

The sparse matrix is created by storing the elements in the array values, and the corresponding column index col_index.

values	10	−2	3	9	3	7	8	7	3	...	13	4	2	−1
col_index	1	5	1	2	6	2	3	4	1	...	6	2	5	6

The row_ptr array tells where the different rows start. The last element in row_ptr is often reserved for storing the number of non-zero elements, z.

row_ptr	1	3	6	9	13	17	19

Let m denote the number of columns, and z denote the number of non-zeros of the matrix, A. The memory requirement for storing A in CRS format is $\mathcal{O}(2z + (m+1))$. For very sparse matrices, this format can reduce the memory requirement considerably.

To create a BCRSMatrix in DUNE, it is needed to set up the sparsity pattern before it can be used. The BCRSMatrix in DUNE offers two ways of doing this:

1. Row-wise: The rows are built up in sequential order. Size of the row and the column indices are defined. The matrix can be used when all the rows have been initialized. This approach is used when the structure of the matrix is known in advance, e.g. finite difference methods.

2. Random: The sparsity pattern is not known in advance, and each row can not be determined in a sequential order. The sparsity pattern require two steps to create the matrix. In the first step all the row sizes are determined, and in the second step the non-zero indices are added.

Both modes offer a special implementation when the number of non-zero elements in the matrix is known in advance.

5.3.3 Sparsity Patterns for the Matrix Blocks

The sparsity patterns for the matrix blocks B, C and Π have to be set up before the blocks can be filled, as discussed in section 5.3.2. The matrix block B is simple, since the choice of basis functions makes B diagonal. The C block is also simple. This is block diagonal with non-zero entries only at indices $(i, 4i)$ for $i = 1, ..., N$. The sparsity patterns for the B and C blocks are set up in row-wise mode.

The sparsity pattern for the Π block is non-trivial. Assume that the internal face F_i is shared by the elements T_k and T_l. Recall that Π_{ij} is given by

$$\Pi_{ij} = \int_{F_i} \psi_j \cdot n_T \, d\Omega,$$

where n_T is the outward pointing normal of F_i in the element T and ψ_j has only support in T. This implies that row i in Π has non-zeros when $j = 4(k-1) + \{1, 2, 3, 4\}$ and $j = 4(l-1) + \{1, 2, 3, 4\}$. The sparsity pattern for the Π block is easiest to set up in a non-sequential order, hence the random mode is used.

Knowing the explicit expressions for the sparsity patterns is important to achieve high efficency when the Schur-complement reductions are performed. In the first version of the implementation, the S matrix in (5.18) was calculated by using matrix-matrix multiplication (MxM) routines for sparse matrices. This approach turned out to be extremely ineffective. The main reason for this is that the general matrix-matrix multiplication implementation can not exploit the fact that it is the matrix is multiplied with the transpose of itself. In addition the matrix-matrix multiplication code can not assume anything about the matrix sparsity patterns in advance.

By using the MxM approach the transformation of the original system of equations used several minutes while the rest of the code used few seconds including solving the resulting system. It was obvious that this way of transforming the equations was not fast enough.

The S matrix consists of two terms, $\Pi B^{-1} \Pi^T$ and $F D^{-1} F^T$. As mentioned above, both B and D is diagonal, so when the sparsity pattern is calculated it is enough to look at $\Pi \, \Pi^T$ and $F F^T$. $F = \Pi B^{-1} C^T$ has non-zeros where ΠC^T has non-zeros. This implies that $F F^T$ has non-zeros where $\Pi (C^T C) \Pi^T$ has. Recall that $D = C B^{-1} C^T$ and is diagonal. This implies that the two terms has identical sparsity patterns.

Let's look at $(\Pi \, \Pi^T)_{ij}$.

$$(\Pi \, \Pi^T)_{ij} = \sum_{k=1}^{4N} \Pi_{ik} \Pi^T_{kj} = \sum_{k=1}^{4N} \Pi_{ik} \Pi_{jk}.$$

Assume that F_i is shared by T_m and T_n and F_j is shared by T_p and T_q. If m or n is equal to p or q, it is possible that S_{ij} is non-zero. This can also be stated in another fashion.

Row i in $\Pi\,\Pi^T$ has entries in the columns corresponding to the global face number of the faces in T_m and T_n. This was implemented using two different maps, `face2elements` and `elementfaces`. `face2elements` maps a global facenumber to two element indices for internal faces, and to one element index for faces located at the boundary. The `elementfaces` map local facenumbers of an element to global facenumbers.

The S matrix, implemented as a `Dune::BCRSMatrix`, was initialized with the following code.

```
//Type alias for the iterator used to create the matrix S
typedef typename BCRSMatrix::CreateIterator Iter;

for(Iter row=S.createbegin(); row!=S.createend(); ++row){
  //Extracts the element indices for the face 'row'
  const int elementP=face2elements[row.index()][0];
  const int elementM=face2elements[row.index()][1];

  //Loops over the 4 faces in the two elements that share
  //the face 'row'
  for (int alpha=0; alpha<4; alpha++){
    //Checks if this face is a non-dirichlet face
    if(elementfaces[elementP][alpha]>=0)
      row.insert( elementfaces[elementP][alpha] );

   //Checks if face, 'row', is located on the boundary,
   //i.e. it is only shared by one element, 'elementP'.
   if (elementM >=0)
      if(elementfaces[elementM][alpha]>=0)
        row.insert( elementfaces[elementM][alpha] );
  }
}
```

By using this approach the generation of the S block takes only a split of a second on the same problem size. This shows how important it is to exploit the structure of the problem. If the transformation would be so costly as the first approach, the transformation would never been used since the cost of transformation was higher than solving the original problem.

5.4 Comments

This chapter introduces Lagrange multipliers, and the size of the resulting system increases. It is also introduced an extra cost by transforming the system into a symmetric positive definite system by performing two Schur-complement reductions, but the transformed system, (5.18), is smaller than the original system, (4.13). For large problems, the extra cost of transforming the system of equations is small compared to the gain in reduced cost by solving the system of equation with an iterative method.

The efficiency of the implementation will be discussed later in the thesis.

Chapter 6

Mixed Hybrid FEM with Cuboid Shaped Elements

In the two previous chapters, tetrahedron shaped elements are used. Arbitrary shaped domains can easily be discretized into a set of tetrahedra with fair coverage of the domains. The upscaling problem makes easily sense on shoe-box shaped domains, and such domains are very common in this context.

The motivation for my thesis is to create upscaling software for the oil industry. The input data is often Eclipse[3] grid files where the permeability data is given in cells. In this thesis the gridfiles are assumed to consist of rectangular cuboids, i.e. stretched cubes with right angles. A cell in the Eclipse grid format can be much more general than this.

The approach used in the previous chapter divided each cell in the cuboid grid into six tetrahedra and the cell's permeability was used in all six tetrahedra. This generates extra degrees of freedom, and makes it unfair to compare the performance with the existing upscaling code.

The cuboid implementation has some limitations compared to the tetrahedron implementation from the previous chapter. The elements are required to be aligned with the coordinate axis, and the shape of the cuboids makes it difficult to describe other domains than cuboid shaped domains with high accuracy.

6.1 Discretization

The domain is now discretized in N cuboid shaped elements. The weak formulation for the mixed hybrid fine element method is unchanged from (5.8). Recall from the previous chapter that it is only necessary to define the velocity function space on each element:

$$RT_0(T) = \left\{ v \in (L^2(T))^3 \;\middle|\; v(x) = a + bx, \text{ where } a, x \in \mathbb{R}^3, \text{ and } b \in \mathbb{R} \right\}.$$

The shapefunctions defined on the reference element associated with the faces $F_1, ..., F_6$, are chosen to be

$$\begin{aligned} \widehat{\psi}_1(x,y,z) &= (1-x)\, e_1, & \widehat{\psi}_2(x,y,z) &= x\, e_1, \\ \widehat{\psi}_3(x,y,z) &= (1-y)\, e_2, & \widehat{\psi}_4(x,y,z) &= y\, e_2, \\ \widehat{\psi}_5(x,y,z) &= (1-z)\, e_3, & \widehat{\psi}_6(x,y,z) &= z\, e_3. \end{aligned} \tag{6.1}$$

[3] Eclipse is an oil and gas reservoir simulator developed by Schlumberger Information Solutions.

From this point it is assumed that all elements in the grid is aligned with the coordinate axes, and all angles in an element are 90 degrees.

Define L as the coordinates of the lower left corner and H as the upper right corner of an element T. The vector $h = H - L$ denotes the grid spacing in each spatial direction. From the assumptions above, the global definition of the shape functions above becomes,

$$\begin{aligned} \psi_1(x,y,z) &= \left(1 - \tfrac{x-L_x}{h_x}\right) e_1, & \psi_2(x,y,z) &= \tfrac{x-L_x}{h_x}\, e_1, \\ \psi_3(x,y,z) &= \left(1 - \tfrac{y-L_y}{h_y}\right) e_2, & \psi_4(x,y,z) &= \tfrac{y-L_y}{h_y}\, e_2, \\ \psi_5(x,y,z) &= \left(1 - \tfrac{z-L_z}{h_z}\right) e_3, & \psi_6(x,y,z) &= \tfrac{z-L_z}{h_z}\, e_3. \end{aligned} \tag{6.2}$$

The shapefunctions has the following property

$$\psi_i \cdot n_{F_i} = \begin{cases} 0 & \text{along } \partial T \setminus F_i \\ 1 & \text{along } F_i. \end{cases}$$

This means that the coefficient corresponding to F_i in the solution vector, is the velocity across F_i. The faces of the cube shaped element is numbered according to the DUNE reference element numbering, except `C++` is 0-indexed. See Figure 6.1.

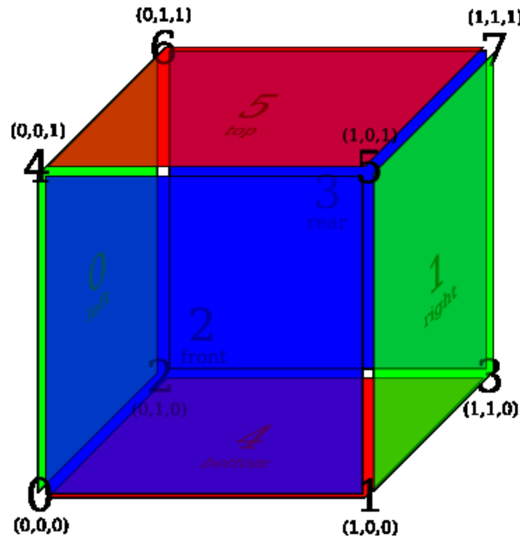


Figure 6.1: The reference cube with the faces and vertices numbered. The figure is from [12].

6.2 Implementation Details

6.2.1 Code Structure

The code for this implementation is listed in section A.2. The code is organized as the implementation for the hybrid method for tetrahedron shaped elements. See section 5.3.1. This implementation is not so general as the previous method. It assumes that the cube shaped elements's faces to be either perpendicular or parallel to the coordinate axes. This is due to the simple mapping used for the global definition of the shape functions. This can be fixed by utilizing a more advanced transformation, but this was not prioritized since the grids used as input are assumed to consist of rectangular cuboids with right angles.

6.2.2 Sparsity Patterns

The local B_T, C_T and Π_T-blocks defined for the cuboid shaped element T are different than for the tetrahedron shaped element discussed in the previous chapter. The dimension of B_T is now 6×6, and the structure is shown in Figure 6.2. The non-zero elements in B_T are located at $B_{i,i}$ and $B_{i,i+2(i \mod 2)-1}$ for $i = 1, ..., 6$. The way the shapefunctions are defined, B_T becomes block diagonal with 3 blocks of dimension 2×2. This results in a global block diagonal matrix. As in the previous chapter it is required to invert the global $6N \times 6N$ matrix B. The computational cost of inverting B is $\mathcal{O}(N)$, since this is the same as inverting $3N$ blocks of size 2.

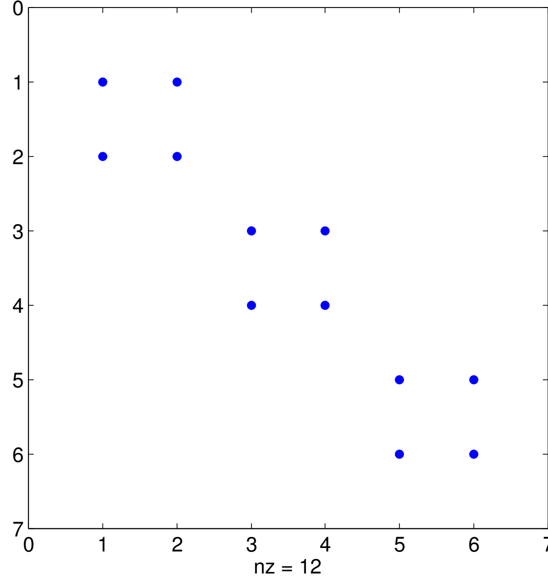


Figure 6.2: The 12 non-zero elements in B_T for a cuboid shaped element.

The local C_T vector is of length 6, and is easy to calculate,

$$C_T = \left(\int_T \operatorname{div}(\psi_i) \, d\Omega \right)_{i=1}^{6} = |T| \left[-\frac{1}{h_x}, \frac{1}{h_x}, -\frac{1}{h_y}, \frac{1}{h_y}, -\frac{1}{h_z}, \frac{1}{h_z} \right].$$

The sparsity patterns for the the global B and C matrix blocks are still easy to set up. The sparsity pattern for the Π-block from section 5.3.3 is also easily extendable to cuboid shaped elements.

Since each row in B^{-1} contains two elements, a matrix-matrix product with B^{-1} potentially connects with two elements. $D = CB^{-1}C^T$ is still diagonal, and each element is given by,

$$D_{ii} = \sum_{\alpha=1}^{6} C_{i,6i+\alpha} \underbrace{\left(B^{-1}_{6i+\alpha,6i+\alpha} C_{i,6i+\alpha} + B^{-1}_{6i+\alpha,6i+\alpha+2(\alpha \bmod 2)-1} C_{i,6i+\alpha+2(\alpha \bmod 2)-1} \right)}_{[B^{-1}C^T]_{6i+\alpha,i}}.$$

The derivations of explicit expressions for the matrix elements are tedious, but the result is much more efficient code compared to using general matrix-matrix multiplication routines for sparse matrices.

6.2.3 Utility Function for Visualizing the Solution

The velocity field is exported for visualization by the `VTKWriter` provided with DUNE. The Visualization Toolkit (VTK) is an open source graphics toolkit. The exported solution can be opened in ParaView, an open-source, multi-platform visualization application built on top of the VTK libraries.

The `VTKWriter` takes an object implementing the `VTKFunction` interface specified in DUNE as an input parameter. This is nothing else than a utility function for how the velocity function, $v(x)$, on the grid is defined. The most important function in this interface is the `evaluate` function. For an element it takes as input a local coordinate, $\widehat{x}$, and returns the corresponding velocity vector. The velocity is calculated by summing the product of the local shapefunctions evaluated in $\widehat{x}$ and the associated solution coeffecients.

The implementation of the visualization function associated with the choice of basis functions is shown below.

```
1  /**********************************************************************
2  * Grid function for the velocity field
3  **********************************************************************/
4  template<class Grid,class Vector>
5  class RT0CubeVelocityFunction :
6      public Dune::VTKWriter<typename Grid::LeafGridView>::VTKFunction {
7   public:
8    typedef typename Grid::ctype ctype;
9    typedef typename Grid::template Codim<0>::Entity Entity;
10   enum{dim=Grid::dimension};
11
12   //Constructor taking the grid, and the velocity solution as input
13   RT0CubeVelocityFunction(const Grid& g, const Vector& v) : grid(g), vel(v)
       {}
14
15   //Number of components in each point
16   virtual int ncomps () const { return 3; }
17
18   //Evaluate component 'comp' on the element 'e' of the velocityfield in
19   //point 'xi' (given in local coordinates)
20   virtual double evaluate (int comp, const Entity& e,
21       const Dune::FieldVector<ctype,dim>& xi) const {
22     double value=0;
23     const int elementIndex=grid.leafIndexSet().index(e);
24
25     //Contribution from the two shape functions which are non-zero
26     value+=vel[6*elementIndex+2*comp]*(1-xi[comp]);
27     value+=vel[6*elementIndex+2*comp+1]*xi[comp];
28
29     return value;
30   }
31
32   virtual std::string name () const {
33     return "velocity";
34   }
35
36   virtual ~RT0CubeVelocityFunction () {}
37
38  private:
39   const Grid& grid;
40   const Vector& vel;
41 };
```

6.2.4 Solving the System of Equations

In the implementation it is chosen to use the conjugate gradient method for solving (5.18). This is implemented in the `dune-istl` module. To achieve more rapid convergence, it is used an algebraic multi grid (AMG) preconditioner on the system of equations. The AMG preconditioner is also part of the `dune-istl`-module.

In solving PDEs numerically, the computational time is often dominated by the time used to solve a linear system of equations. The choice of the best solver is problem dependent, but algebraic multigrid methods are known to be efficient on systems arising from elliptic PDEs [13]. Later in the thesis, the efficiency of this approach will be compared with other solvers available.

Even with an efficient linear algebra solver as the combination CG and AMG, the total computational time is dominated by the time used solving the system of equations.

6.3 Installation and Usage

The implementation is available at `http://folk.ntnu.no/arne/upscaling.tar.gz`, licensed under the GPL license version 2 or later. The implementation is created as a DUNE module and have dependencies to the DUNE core modules `dune-common`, `dune-grid` and `dune-istl`. DUNE provides it's own build-system, and the easiest way to integrate with this is to create a DUNE module. The buildsystem is based on `autotools` from GNU.

To configure and build the `upscaling` module, it is necessary to download and extract the `upscaling` module and the DUNE core modules[4] of version 1.2 in the same base directory. The result is the following directory structure:

`base/`

- `dune-common-1.2`
- `dune-grid-1.2`
- `dune-istl-1.2`
- `upscaling`

For high performance, it is recommended to have `SuperLU`[5] installed. It is also strongly recommended to have a newer compiler with template function optimizations, e.g. `gcc-4.3` or newer. The configuration file, `config.opts`, for utilizing `SuperLU` and turning on highest level of optimizations, looks like this

```
CONFIGURE_FLAGS="CXXFLAGS='-O3' --with-superlu=/path/to/SuperLU
--with-superlu-lib=superlu.a"
```

For Mac OS X based system, the compiler flag `'-O3'` can be replaced by `'-fast'` for better performance.

[4]These can be downloaded from DUNE's webpage: `http://wwww.dune-project.org`

[5]Implementation of LU-factorization for sparse systems. Used in AMG to solve the coarsest level's system of equations. `http://crd.lbl.gov/~xiaoye/SuperLU/`

All modules are configured and built by invoking the command from `base`

```
./dune-common-1.2/bin/dunecontrol --opts=config.opts all
```

The `upscaling`-executable file is now found in `base/upscaling/src` and has the following syntax:

```
./upscaling eclipsefile [writeVTK=0]
```

The `eclipsefile` variable is the Eclipse file and the only keywords necessary are `SPECGRID`, `COORD` and `ZCORN` specifying the grid, and `PERMX` specifying the permeability field. The optional `writeVTK` variable is a boolean implemented as 0 and 1 whether to export the solution for visualization in VTK-format or not. The output is written to the file `upscaling.vtk`, which can be visualized with use of Paraview.

The input Eclipse file for the model shown in Figure 3.3 is listed in section A.1.

Chapter 7

Algebraic Multigrid (AMG)

Algebraic multigrid is used in solving the linear system of equations in this thesis. Algebraic multigrid methods are known to be an efficient technique to solve large systems of linear equations arising from elliptic PDEs [13].

7.1 What is AMG?

The motivation is to solve the symmetric positive definite system,

$$Ax = b, \quad \text{where } A \in \mathbb{R}^{N\times N} \text{ and } x, b \in \mathbb{R}^N,$$

The idea of algebraic multigrid comes from the standard multigrid methods used in solving PDEs. The development of multigrid methods started with a detailed analysis of classic iterative methods for solving $Ax = b$, like the Jacobi and Gauss-Seidel method. These methods tend to reduce the high frequency error components efficiently, while the low frequency error components are reduced slowly. Instead of reducing the iteration error, $e^{(i)} = x - x^{(i)}$, they only smooth the error. They are often referred to as smoothers.

The error function $e^{(i)}$ can be expressed as a superposition of sine waves of different wavelengths. Multigrid theory states that the smoother reduces well those components of the error whose wavelength is short with respect to the grid width while it is unable to reduce long wavelength components of the error. The iterative solver alone is therefore unable to achieve rapid convergence.

For any multigrid method it is necessary to define a hierarchy of grids as illustrated in Figure 7.1, transfer operators between the grid levels, a smoothing operator, coarse-grid version of the fine-grid operator and a solver for the coarsest grid [14]. In standard or geometric multigrid methods, the unknown variables x_i are defined at known spatial locations or grid points in a fine grid. A subset of the points is selected as the coarse grid, and a subset of x_i is used to represent the solution at the coarse grid.

For algebraic multigrid methods, the physical locations of the nodes in the grid are unknown. We seek a subset of the variables x_i to serve as the coarse-grid unknowns. The grid points are defined to be the indices $\{1, 2, ..., N\}$. Having defined the grid points, the connections within the grid are determined by the undirected adjacency graph of the matrix A. A can be represented as a graph with N nodes where an edge between node i and j represents a non-zero a_{ij} or a_{ji}. The grid information in algebraic multigrid methods are entirely determined by the matrix A.

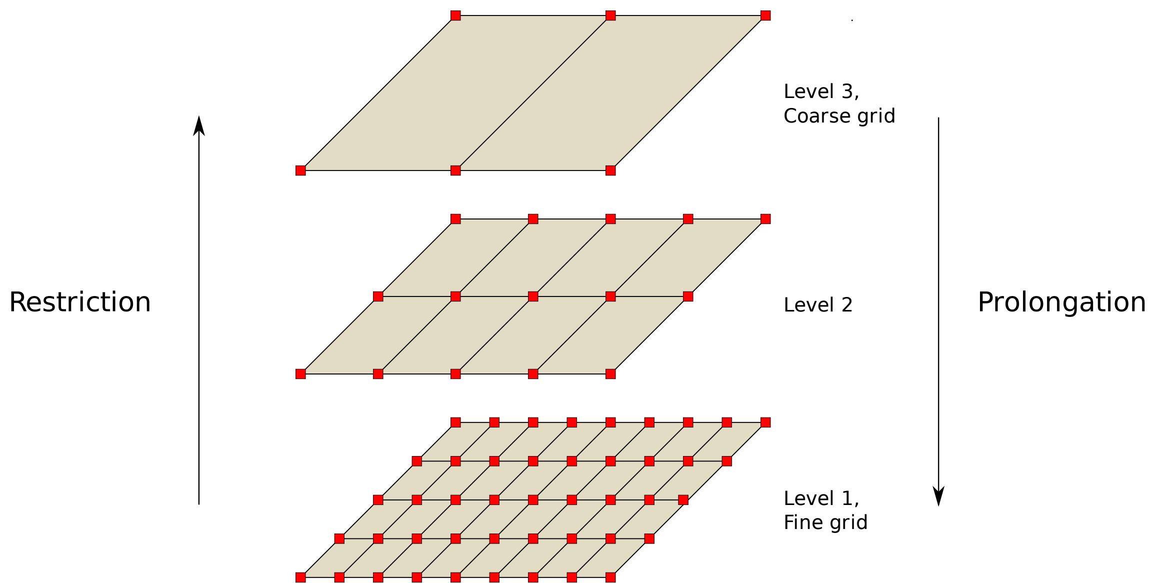


Figure 7.1: The different grid levels and interpretation of the prolongation and restriction operators on a 2D regular grid.

Multigrid methods only use a number of iterations of the smoother that is sufficient to reduce the short wavelength components. Since a wavelength that is long with respect to the fine grid is short with respect to some coarser grid, traversal of the grid hierarchy and application of the smoother reduces all components of the error function. To achieve convergence, it is required that an exact solution is computed at the coarsest grid level. The number of unknowns at the coarsest level is usually a small number, and a direct solver like complete LU-factorization can be used.

The AMG method was intended to be used as a standalone iterative solver, but AMG also works as a good preconditioner. The solver used in this thesis is the conjugate gradient method, with DUNE's implementation of algebraic multigrid based on aggregation as preconditioner. This approach is compared with the SAMG solver developed by Fraunhofer SCAI. The largest algorithmic difference is that DUNE is aggregation based while SAMG is not. Aggregation means that the fine scale variables interpolates from just one coarse scale variable, even though the fine scale variables are connected with several coarse scale variables [13]. The advantage is less memory usage at the cost of reduced performance. This behaviour is actually seen in the results section. The memory usage of the DUNE solver is about half of the SAMG solver, but the SAMG solver is faster.

7.2 Two-Level Grid Cycle

This section describes a two-level grid cycle in a multigrid method. The algebraic multigrid differs from geometric multigrid methods in the interpretation of the "grid", and how the grid transfer operators are defined. The terminology used describing the methods is the same.

Assume that we want to solve the Poisson equation $-\Delta u = f$ discretized on the unit square with the five point stencil. Then the eigenvalues of the discrete Laplace operator

A are known. The systems of equations given on the fine grid and coarse scale are given by,

$$Au = b \quad \text{and} \quad A_c u_c = b_c,$$

respectively. u_c may be computed, and prolongated onto the fine grid:

$$u^{(1)} = P u_c,$$

where P is the prolongation operator. $u^{(1)}$ can now be used as an initial approximation for the fine scale system, i.e. the problem can be formulated as

$$A(u^{(1)} + e) = b, \quad \text{or} \quad Ae = b - APA_c^{-1} b_c = r^{(1)}.$$

This process can be repeated by restricting the residual to the coarse grid,

$$r_c = R r^{(1)},$$

where R is the restriction operator. Then finding the correction on the coarse grid,

$$e_c^{(2)} = A_c^{-1} r_c,$$

and prolongate the coarse-grid correction onto the fine grid

$$\begin{aligned} e^{(2)} &= P e_c^{(2)} \\ u^{(2)} &= u^{(1)} + e^{(2)}. \end{aligned} \tag{7.1}$$

This is called the coarse grid correction step of the multigrid algorithm.

To see if this works, let's assume that $b_c = Rb$. One has that

$$\begin{aligned} u - u^{(1)} &= u - PA_c^{-1} Rb \\ &= (I - PA_c^{-1} RA) u \\ u - u^{(2)} &= (I - PA_c^{-1} RA)^2 u, \end{aligned} \tag{7.2}$$

and so on. Similary for the residual,

$$\begin{aligned} r^{(1)} &= b - APA_c^{-1} Rb \\ &= (I - APA_c^{-1} R) r^{(0)}. \end{aligned} \tag{7.3}$$

It is possible to show that

$$\rho(I - APA_c^{-1} R) = \rho(I - PA_c^{-1} RA) \geq 1,$$

i.e. the correction will not converge if applied many times. The solution for this is to apply a smoother. This will smoothen the oscillatory error $u - u^{(1)}$ and the residual $r^{(1)}$. A simple iterative method that smooths the high frequency components of the error, can be formulated as

$$u^{(1)} = u^{(0)} + M^{-1} r^{(0)}.$$

For example, Gauss-Seidel and Successive Over Relaxation (SOR) work as smoothers. The error estimates for the iterative methods becomes

$$\begin{aligned} u - u^{(1)} &= (I - M^{-1} A)(u - u^{(0)}) \\ r^{(1)} &= (I - AM^{-1}) r^{(0)}. \end{aligned} \tag{7.4}$$

This is not efficient alone, since $\rho(I - M^{-1} A) \approx 1$, so it converges slowly, but the high-frequency components are reduced efficiently in few iterations. The steps in one two-level grid cycle are

- Presmooth: Reduce the high frequency error coefficients on the fine scale grid
- Calculate the coarse grid correction, e_c.
- Prolongate the coarse grid correction onto the fine grid
- Postsmooth: Reduce the high frequency error coefficients on the fine scale grid

According to [15] it can be shown that $\rho\left((I - M^{-1}A)(I - PA_c^{-1}RA)(I - M^{-1}A)\right) \ll 1$. This is the spectral radius of the iteration matrix for the multigrid cycle, and the result tells that the multigrid method reduces all error components efficiently.

The two-level method can easily be extended to several levels, by recursively calling the two-level method instead of the exact solver at the coarsest level. It is often desirable to have a sufficiently low number of unknowns on the coarsest level, since this is solved with use of an exact solver. Solving the coarsest system exact can be expensive if it is too large.

In [13] it is emphasised that the speed of convergence strongly depends on the interplay between relaxation (smoothening) and interpolation (prolongation and restriction).

This section says nothing about how the prolongation and restriction operator should be constructed. This is described in detail in [13].

Chapter 8

Numerical Results

This chapter contains discussion regarding verification and efficiency of the implementations. The result from a permeability upscaling of a core sample is also discussed.

8.1 Verification of the Implementations

The implementations were tested on the simplest problem possible of this type. This equals a constant permeability field K equal to 1 in the domain Ω. The problem considered is

$$\begin{aligned} v = -\nabla p &\quad \text{in } \Omega = (0,1)^3, \\ \nabla \cdot v = 0 &\quad \text{in } \Omega, \\ p = 0 &\quad \text{on } \Gamma_{D_0} = \{z|\ z = 0\}, \\ p = 1 &\quad \text{on } \Gamma_{D_1} = \{z|\ z = 1\}, \\ v \cdot n = 0 &\quad \text{on } \Gamma_N = \partial\Omega \setminus \{\Gamma_{D_0} \cap \Gamma_{D_1}\}. \end{aligned} \tag{8.1}$$

It is easy to verify that a solution to (8.1) is

$$p(x, y, z) = z \quad \text{and} \quad v(x, y, z) = [0,\ 0,\ -1]^T.$$

This test worked as a first hand check whether the system of equations were set up and solved correctly. A numerical solution is shown in Figure 8.1. The pressure looks to be independent of the x and y-coordinates as expected. The analytical solution to the test problem is linear and it is not possible to use this example to analyze the convergence rate of this mixed finite element method since the numerical solution is expected to be exact to the magnitude of machine precision.

The maximum componentwise error is calculated as

$$\|p - p_{\text{exact}}\|_\infty = \max_{k=1,\dots,N} |p_k - p_{\text{exact}}(\text{centroid}(T_k))| = \max_{k=1,\dots,N} |p_k - \text{centroid}_z(T_k)|.$$

This method is expected to give exact solution on a simple problem like this. With $N = 6 \cdot 2^3 = 24$ elements, the maximum componentwise error is in the range of the convergence criterion (error of magnitude 10^{-9}).

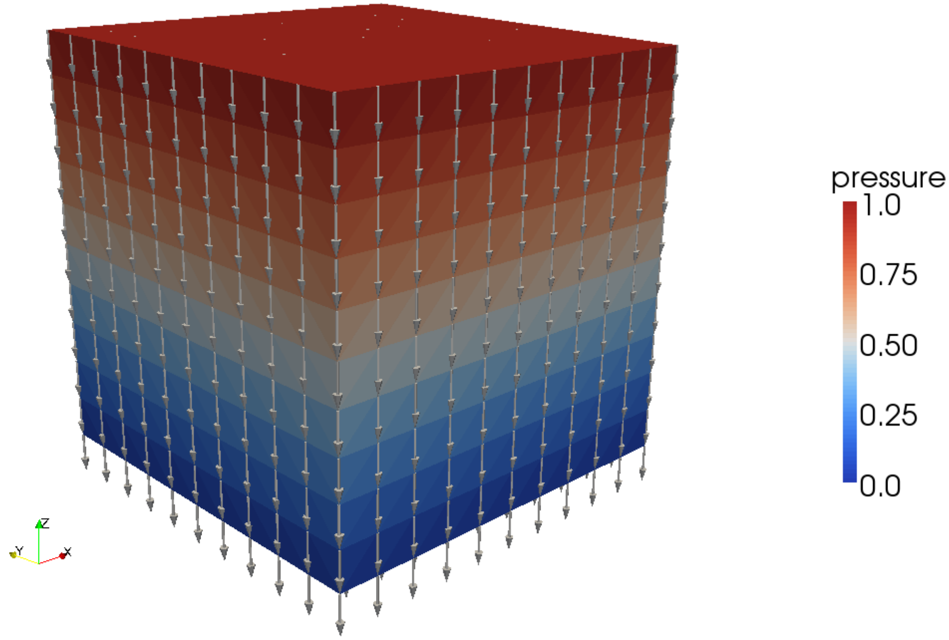


Figure 8.1: The solution to (8.1). The coloring is the pressure p, and the arrows denote the velocity field $v = -\nabla p$. Ω is discretized into $6 \cdot 10^3$ tetrahedra.

8.1.1 Error Analysis

The cuboid implementation is tested on a synthetic case. The solution to the problem

$$\begin{aligned} v = -K(x)\nabla p \quad & \text{in } \Omega = (0,1)^3, \\ \nabla \cdot v = 0 \quad & \text{in } \Omega, \\ p = 0 \quad & \text{on } \Gamma_{D_0} = \{z \mid z = 0\}, \\ p = 1 \quad & \text{on } \Gamma_{D_1} = \{z \mid z = 1\}, \\ v \cdot n = 0 \quad & \text{on } \Gamma_N = \partial\Omega \setminus \{\Gamma_{D_0} \cap \Gamma_{D_1}\}. \end{aligned} \tag{8.2}$$

is found on a fine grid with grid size h. The permeability field, K, is defined to be

$$K(x,y,z) = \begin{cases} 10 & \text{for } |x-0.5| \geq 0.3 \text{ or } |y-0.5| \geq 0.3 \\ 1 & \text{elsewhere.} \end{cases} \tag{8.3}$$

The permeability field can be described as a low-perm cuboid inscribed in a high-perm medium. See Figure 8.2.

The solution corresponding to the fine scale grid is used as an approximation of the exact solution. A numerical solution to (8.2) is shown in Figure 8.2.

According to error analysis in [10], assuming a constant permeability field, the error of the mixed method,

$$\|v - v_h\|_{H^{\mathrm{div}}(\Omega)} + \|p - p_h\|_{L^2(\Omega)} \leq Ch\|v\|_{H^2(\Omega)}.$$

The convergence rate of the error can be approximated by finding the solution of grids

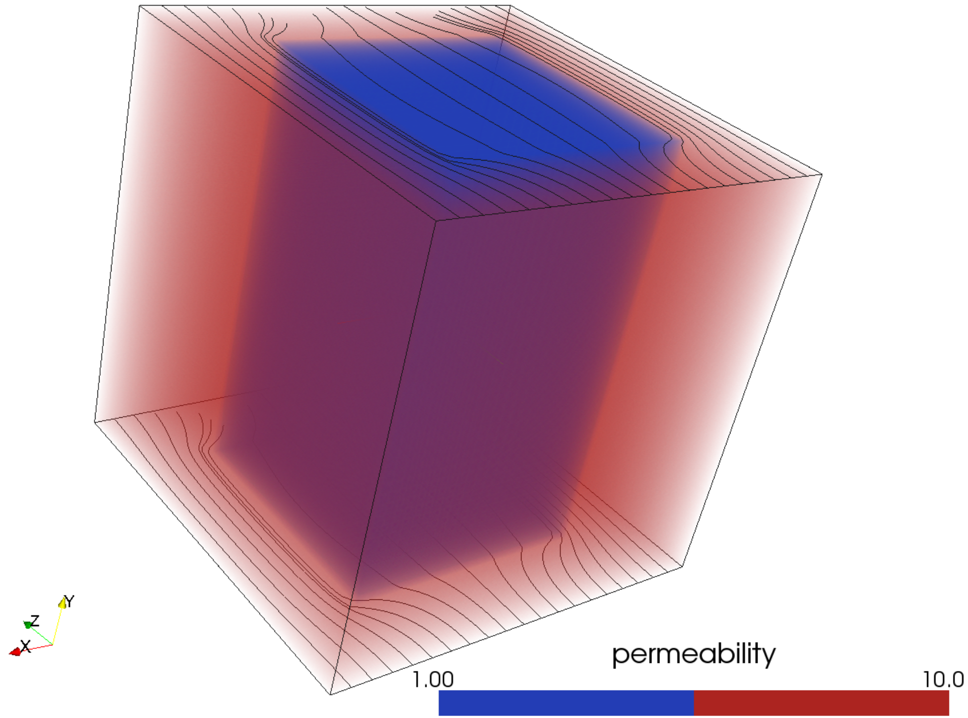


Figure 8.2: The permeability field and the numerical solution to (8.2), represented by streamlines calculated in Paraview. The asymmetry of the streamlines is from Paraview and not the numerical solution.

with grid size $H > h$. The error in the pressure variable can be approximated by

$$\begin{aligned}
\|p - p_H\|_{L^2(\Omega)} &= \|p - p_h + p_h - p_H\|_{L^2(\Omega)} \\
&\leq \|p_h - p_H\|_{L^2(\Omega)} + \underbrace{\|p - p_h\|_{L^2(\Omega)}}_{\mathcal{O}(h)} \\
&\approx \|p_h - p_H\|_{L^2(\Omega)} = \left(\sum_{k=1}^{N_h} h^3 \left((p_h)_k - (P_H^h p_H)_k \right)^2 \right)^{1/2},
\end{aligned} \tag{8.4}$$

where P_H^h prolongates the solution vector from the grid with N_H elements to a solution vector of length N_h.

The error in the pressure variable was calculated as described in (8.4) for different mesh sizes H. The solution on a uniform grid with $N_h = (160)^3 = 4\,096\,000$ elements was used as an approximation to the exact solutions. The error is shown in Figure 8.4. Based on linear regression on the following regression model,

$$\log \|p_h - p_H\|_{L^2(\Omega)} = C + q \log H,$$

the order of convergence, q, was estimated to 1.06. This result coincides with the theoretical bound for the error.

A similar error estimate can be calculated for the velocity field,

$$\begin{aligned} \|v - v_H\|^2_{H^{\mathrm{div}}(\Omega)} &\approx \int_\Omega |v_h(x) - v_H(x)|^2 + (\mathrm{div}(v_h(x) - v_H(x)))^2 \; d\Omega \\ &= \sum_{k=1}^{N_h} \int_{T_k} \underbrace{\left|v_h^k(x) - v_H^k(x)\right|^2 + \left(\mathrm{div}(v_h^k(x) - v_H^k(x))\right)^2}_{e^k(x)} \; d\Omega, \end{aligned}$$

where v_h and v_H denotes the approximate velocity solution on grids with element sizes h and H respectively and $h < H$. The error is calculated by defining an error function on each fine scale element T_k, $e^k(x)$, and adding up the contributions from all elements in the fine scale grid.

The integral of the error function, $e^k(x)$, is calculated by applying a numerical quadrature (Gauss-Legendre) with a sufficient high order such that the quadrature error becomes neglectable compared with the discretization error. The sets of the quadrature points, $\{\xi_\alpha\}_{\alpha=1}^m$, and the corresponding weights, $\{w_\alpha\}_{\alpha=1}^m$, defines the quadrature rule. The integral can be approximated by

$$\begin{aligned} \int_{T_k} e^k(x) \, d\Omega &= \int_{\widehat{T}} \underbrace{e^k(\, g^k(\hat{x})\,)}_{\hat{e}^k(\hat{x})} |J(\xi)| \, d\xi \\ &\approx h^3 \sum_{\alpha=1}^m w_\alpha \hat{e}^k(\xi_\alpha) \end{aligned}$$

where $g^k(\cdot)$ is the local-to-global map and $|J(\cdot)|$ denotes the determinant of the Jacobian of the inverse map. This is necessary due to transformation of the variables in the integral. See Figure 8.3. For the simple grid in this analysis, $|J(\xi)| = h^3$ for all elements.

The error estimates for different grid sizes is shown in Figure 8.4. Regression estimates the convergence rate for the velocity to be $\mathcal{O}(h^{0.83})$. This is a bit slower than the expected theoretical bound which is $\mathcal{O}(h)$. A reason for the slower convergence rate is most likely related to the inhomogeneous permeability field. The permeability field has some jumps which could lead to singularities in the solution and affect the convergence rate.

In Figure 8.5 a cross-section of the spatial error distribution is shown. There are large error estimates at the corners of the inscribed low-perm cuboid. By local refinement of the grid in these areas, the convergence rate may increase.

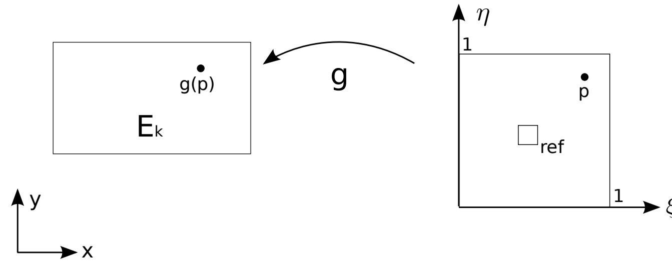


Figure 8.3: Map, g, from the reference element $\square_{ref}$ to the element E_k.

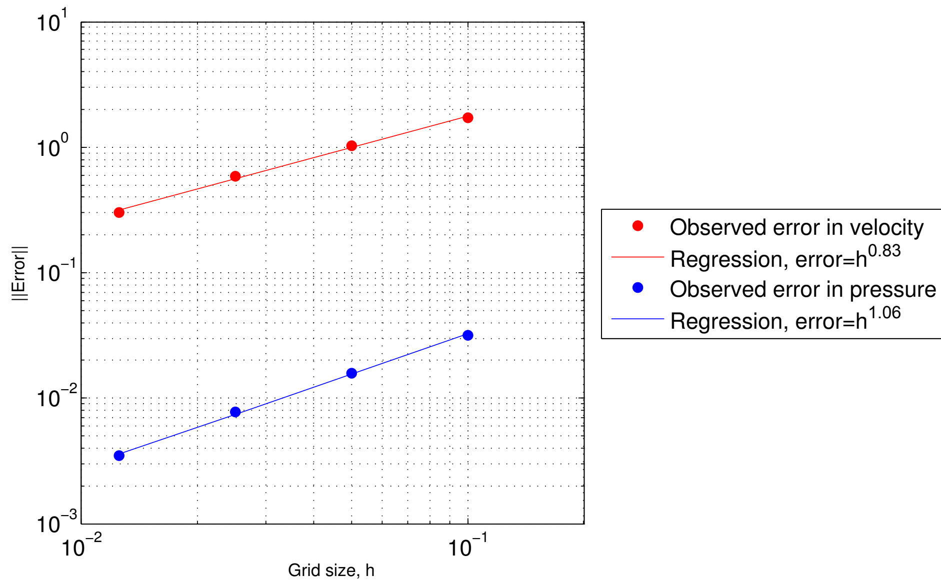


Figure 8.4: The observed error estimates vs. the grid size. The observed errors in the velocity variable, $\|v - v_h\|_{H^{\text{div}}(\Omega)}$ is denoted by red dots. Regression estimates the error to be proportional to $h^{0.83}$. The error in the pressure variable, $\|p - p_h\|_{L^2(\Omega)}$, is denoted by blue dots. Regression estimates the error to be proportional to $h^{1.06}$.

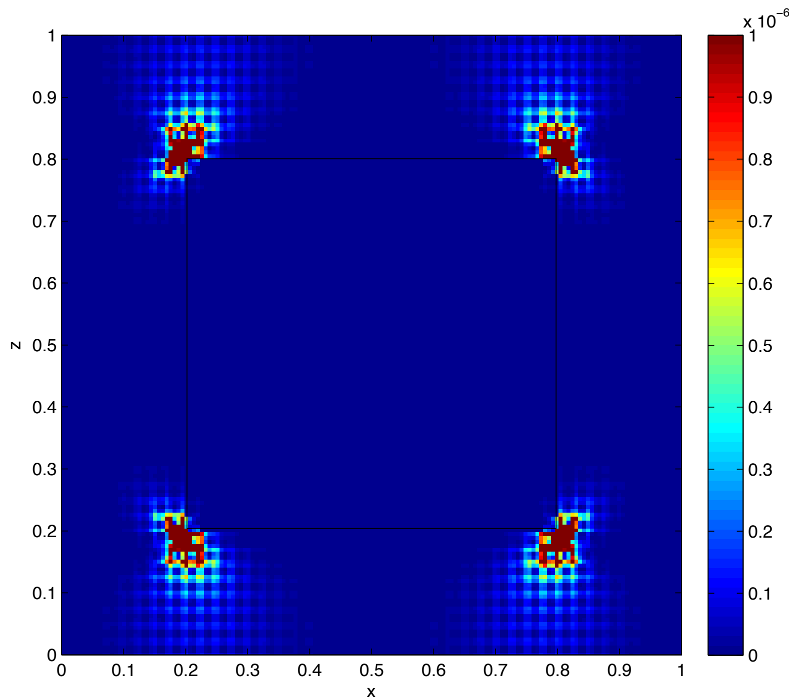


Figure 8.5: Error estimate of the velocity variable in a cross-section $(y = 0.5)$ of the model shown in Figure 8.2. The error at the corners of the inscribed low-perm cuboid is very large compared with the error in the rest of the domain.

8.1.2 Tetrahedron Implementation

The implementations based on tetrahedra shaped elements described in chapter 4 and 5 should be independent of the geometry. In these implementations the problem must be specified in terms of boundary conditions and the permeability field in the domain.

A pipe shaped domain with bends and a homogeneous permeability field is used to verify that the implementation is flexible with respect to the geometry. The difference in pressure between the two end surfaces of the pipe is equal to one. The solution of the problem can be shown in Figure 8.6. The model was downloaded from INRIA's 3D mesh database[6] and can only be used for research or non-commercial use.

The solution shows that the velocity is larger at the inner curves. This behaviour can be explained by a larger pressure drop pr. distance in these areas. See Figure 8.7. By Darcy's law the velocity is proportional to the gradient of the pressure, and explains the higher velocities at the inner curves.

The correctness of the implementation is verified by calculating the upscaled permeability tensor on the two-layered model with 8 cuboids shown in Figure 3.3. Every cuboid is divided into 6 tetrahedra. The upscaled permeability tensor calculated by this implementation is exactly the same as the analytical solution found in section 3.2.

8.1.3 Cuboid Implementation

The correctness of the implementation based on cuboid shaped elements is verified by the error analysis in section 8.1.1, and by comparing the upscaled permeability tensor of a core sample with the tensor found with Sintef's implementation. The two implementations gives the exact same permeability tensor and the implementation is assumed to be correctly implemented.

[6] INRIA GAMMA's 3D mesh database, `http://www-c.inria.fr/Eric.Saltel/`

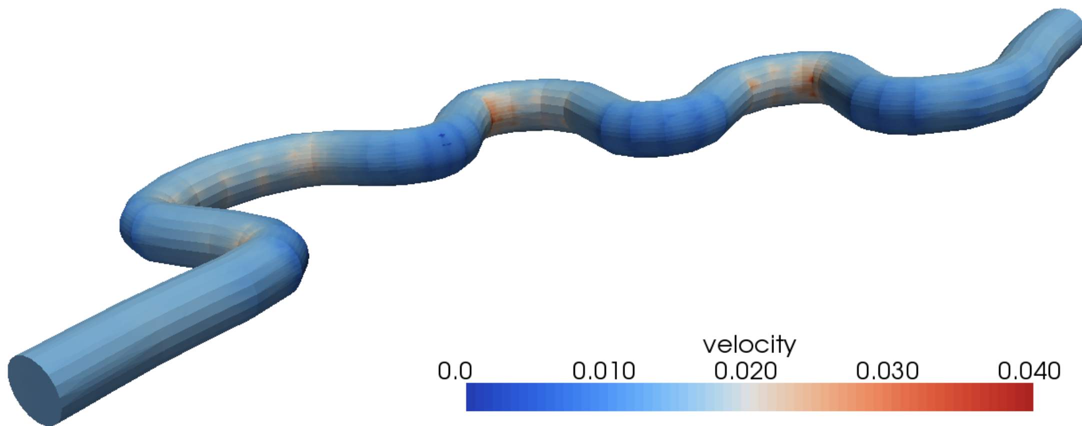


Figure 8.6: The flow problem solved on a more complex grid based on tetrahedra. The boundary conditions are $p = 1$ on the end surface to the left and $p = 0$ on the end surface to the right. On the rest of the boundary, the no-flow condition is applied.

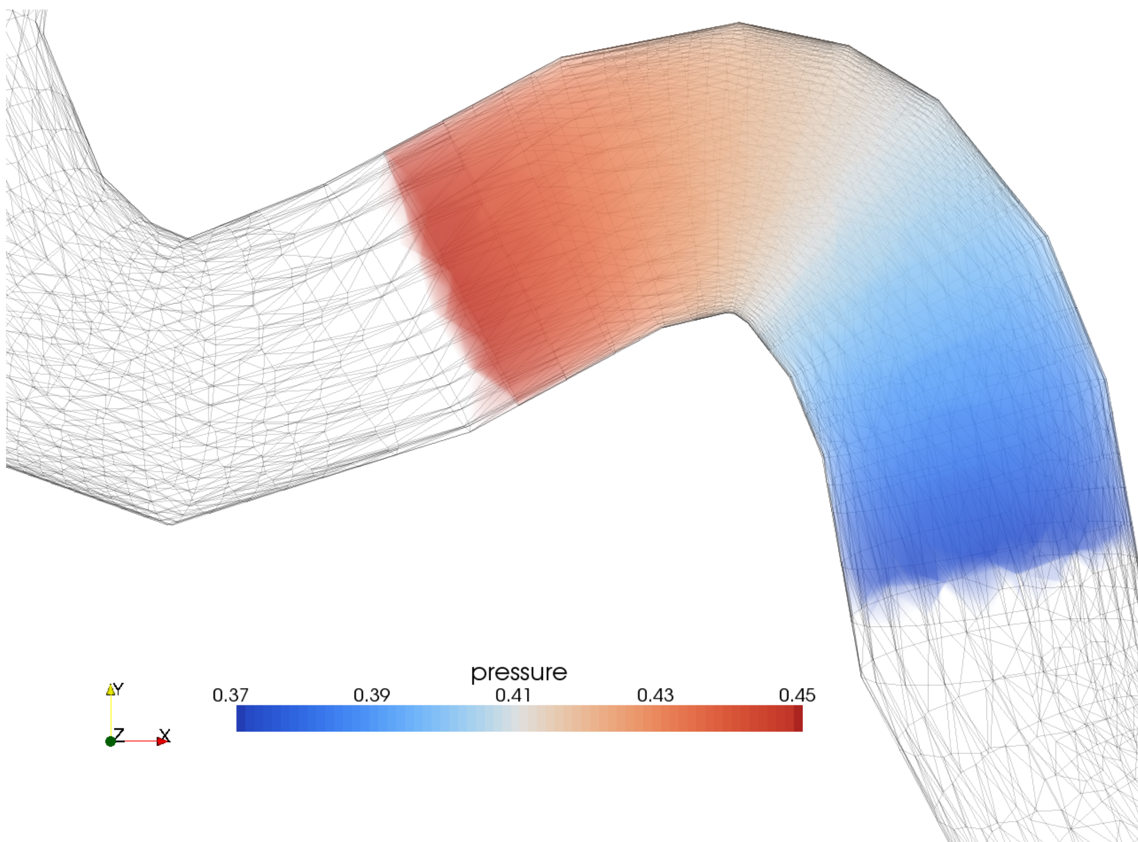


Figure 8.7: The pressure through a bend on the model above. This together with Darcy's law explains why the velocity is higher through an inner curve.

8.2 Upscaled Permeability of a Core Sample

The implementation based on cuboid shaped elements was used on a model with permeability data coming from a laboratory study of a core sample. The dimension of the homogeneous grid is $100 \times 100 \times 11$ blocks. The permeability field and the numerical solution is shown in Figure 8.8. The solution shows that the flow goes through the high permeable areas of the plug in the direction of the pressure drop, as expected. The figure is generated in Paraview.

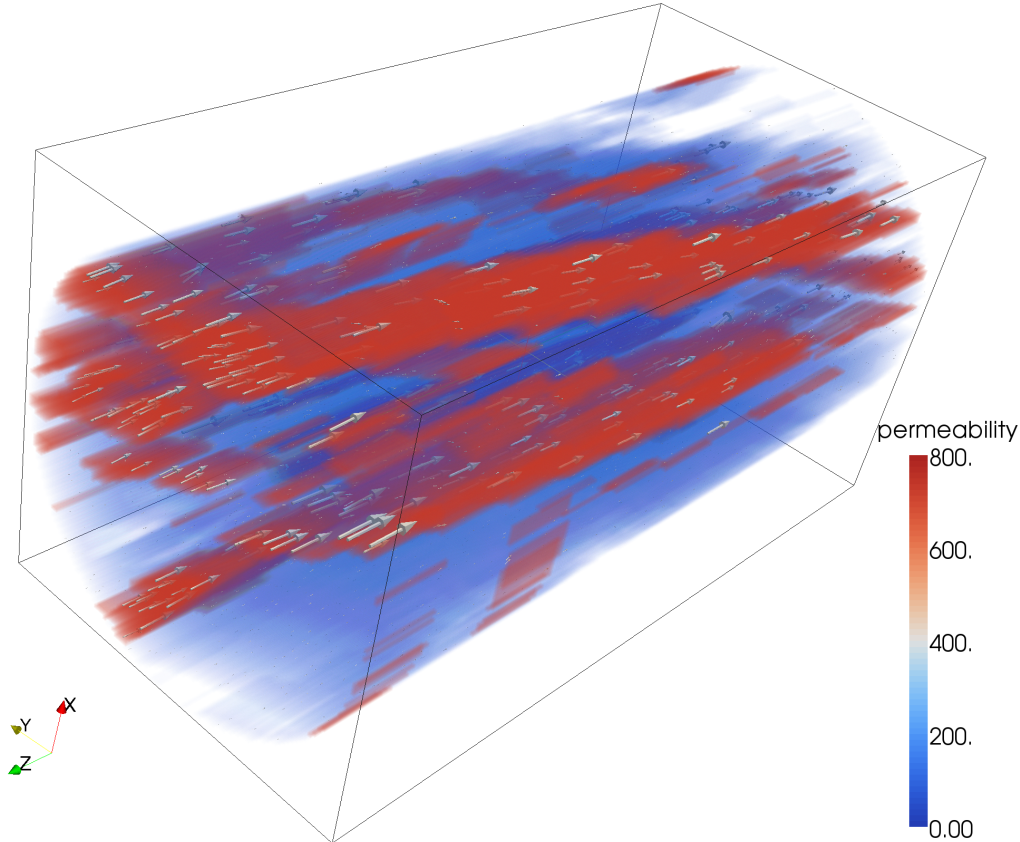


Figure 8.8: The permeability field and the velocity solution of (4.2) with fixed boundary conditions and pressure drop in the negative z-direction.

Fixed boundary conditions are applied, i.e. no flow through other faces than the faces with normal vector parallel to the pressure drop direction. This yields a diagonal upscaled permeability tensor,

$$\widetilde{K} = \begin{bmatrix} 3.02162 \cdot 10^{-8} & 0 & 0 \\ 0 & 3.02103 \cdot 10^{-8} & 0 \\ 0 & 0 & 114.738 \end{bmatrix}.$$

The upscaled horizontal permeabilities are of magnitude 10^{-8}. This is because the cylindrical shaped plug is inscribed in a cuboid shaped box. All elements outside the cylinder have permeability set to 10^{-9} which in practice means that no flow goes through this area. This is the exact same result as the upscaling code developed by Sintef gives. Their

implementation utilizes a mimetic finite difference method which is more flexible with the shape of the elements. On a regular grid the mimetic finite difference method and the mixed finite element with lowest order Raviart-Thomas elements are equivalent [7].

8.3 Efficiency

The implementation based on cuboid shaped elements generate system of equations of same size as the implementation from Sintef. Since the generated problem is of the same size, it is possible to compare results, time usage and memory requirement of the solvers.

Making the code efficient has been a long process. The plug showed in Figure 8.8 with 110 000 elements and 322 200 unknowns (pressure at the interior faces) was used as a test case. The run-time of this program was initially several hundreds times slower than the program implemented by Sintef. First of all the cuboid shaped elements needed to be implemented so the system of equations could be comparable. The implementation based on tetrahedra shaped elements, divides each cuboid in the model into tetrahedra, and the system matrix increases accordingly. By implementing cuboid shaped elements, the problem size became identical.

The second dramatic improvement was to abandon the matrix-matrix multiplication approach used in creating the Schur-complement reduction matrix (discussed in section 5.3.3). These improvements made the implementation approx. 4 times slower than the Sintef implementation. From this point, the computational time is dominated by the time used in the linear solver, and not by the time used to transform the system of equations. Functionality for solving the linear system of equations is provided within DUNE.

The source code and the results of a time measurement was posted on DUNE's mailinglist. Some of the developers of DUNE looked into the code and made proposals of fine-tuning the setup of the AMG preconditioner for this problem. This reduced the solution time to a factor of around 2.3 compared with SAMG. The best proposal was to use the direct solver, `SuperLU`, to solve the system on the coarsest level.

The latest improvement was to use a newer compiler than `gcc-3.4.6`. The later versions of `gcc` apply more optimization for template parameterized functions and this benefit the heavily templatized DUNE framework. The final implementation uses now 58% longer time than SAMG. The different timing results can be showed in Table 8.1.

From the table, it can be seen that just by upgrading the compiler, it is possible to reduce the computation time by approx 30%(!) for all problem sizes. For the largest problem size, the time is reduced by over 50%. This is explained by new optimizations of template parametrized functions and classes introduced in the later `gcc` compilers. These have a great impact on heavily templated code such as code based on DUNE. The SAMG solver is implemented in Fortran, and requires the matrix and right hand side to be in a specific data structure. Most likely it is not possible to optimize the SAMG solver considerably by development of newer compilers, since the information about the data representation of the matrices and vectors could be (and probably was) exploited when the solver was developed.

It is also possible to see that the AMG approach in DUNE uses more iterations than the SAMG solver, and the number seem to grow with the problem size. It may be possible to tune the parameters in the AMG preconditioner for this problem to reduce the number of iterations, but there are algorithmic differences between the methods. According to the developers of DUNE, the iteration count is expected to increase with the problem size for this type of problem. A great advantage with the implementation based on DUNE

Table 8.1: Different timing results depending on which program used and problem size. Setup denotes the time used to set up the grid hierarchy.

	Dune, `gcc-3.4.6`			**Dune**, `gcc-4.3.3`			**SAMG**		
	Setup	**Total**	**Iter.**	**Setup**	**Total**	**Iter.**	**Setup**	**Total**	**Cycles**
N	[sec]	[sec]	[-]	[sec]	[sec]	[-]	[sec]	[sec]	[-]
13 240	0.25	0.53	15	0.14	0.34	15	0.12	0.36	17
29 460	0.60	1.33	16	0.32	1.00	16	0.28	0.78	17
81 100	1.73	4.21	20	0.95	2.68	20	0.88	2.27	17
116 520	2.52	6.57	23	1.41	4.43	23	1.25	3.30	17
206 560	4.51	13.04	26	2.54	8.91	26	2.34	6.09	17
261 180	5.73	16.84	27	3.23	11.31	27	3.05	7.78	17
322 200	7.12	21.46	29	4.00	15.13	29	3.78	9.55	17
3 042 424	75.61	418.23	50	38.91	208.92	50	39.72	100.72	18

is that the memory usage is only 50% of Sintef's implementation for the largest problem investigated (approx. $3 \cdot 10^6$ unknowns). The memory usage in Sintef's implementation is dominated by the SAMG solver.

The stop criterion used in these two linear solvers are not exactly the same, since the solvers are not based on identical alghorithms. The timing results is not directly comparable, since one of the solver may solve the system with higher accuracy than the other. What actually can be seen in Figure 8.9 is that the DUNE solver has higher complexity than the SAMG solver. One explanation for this, is that the number of iterations used by the conjugate gradient method increases with the problem size. The SAMG solver uses one extra iteration when the number of unknowns increases from 322 200 to 3 042 424, a factor of almost 10, while CG uses 21 extra iterations for the same increase in number of unknowns.

8.3.1 Comparing DUNE's AMG with other Alternatives

The system of equations from the synthetic case discussed in section 8.1.1 was used to compare the efficiency of DUNE's AMG preconditioner with other alternatives. The result of this comparison can be seen in Figure 8.10. The direct solver, `SuperLU` has a higher complexity than the iterative methods, and is also verified by this analysis. The `SuperLU` solver is not suitable for large problems, but the advantage with the direct solver is that it is not dependent of a symmetric, positive-definite matrix as the conjugate gradient method.

The two preconditioned conjugate gradient solvers with respectively symmetric successive over-relaxation and incomplete LU factorization used as preconditioner have comparable performance, but the AMG solver accelerated with the conjugate gradient method is superior to the other solvers. For the largest problem investigated with $1.27 \cdot 10^6$ unknowns it uses 33 seconds, while the next best alternative, CG with ILU0 uses 95 seconds, and the difference increase accordingly with the problem size.

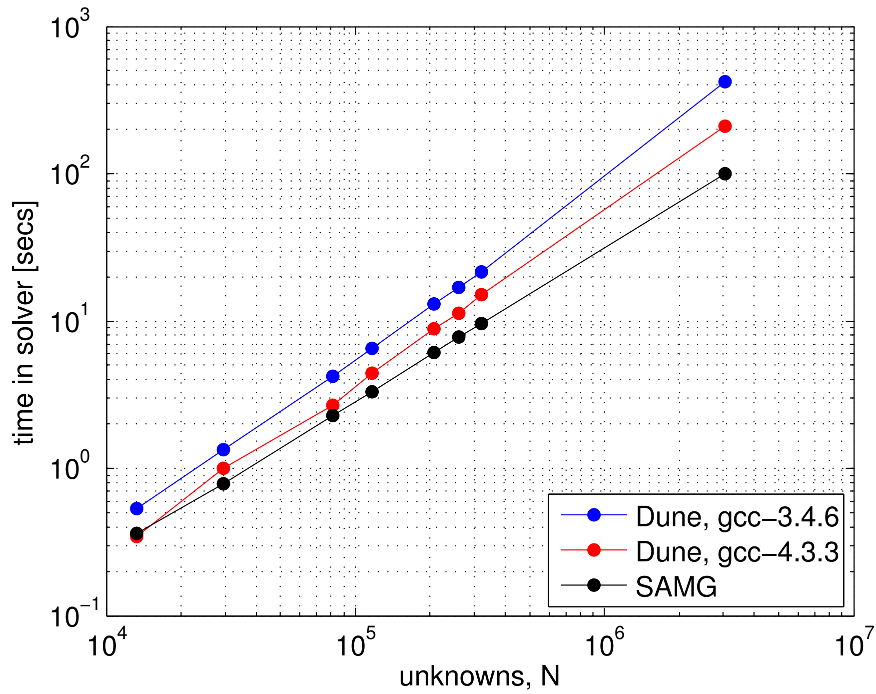


Figure 8.9: Time used by the solver in the three cases with respect to the problem size

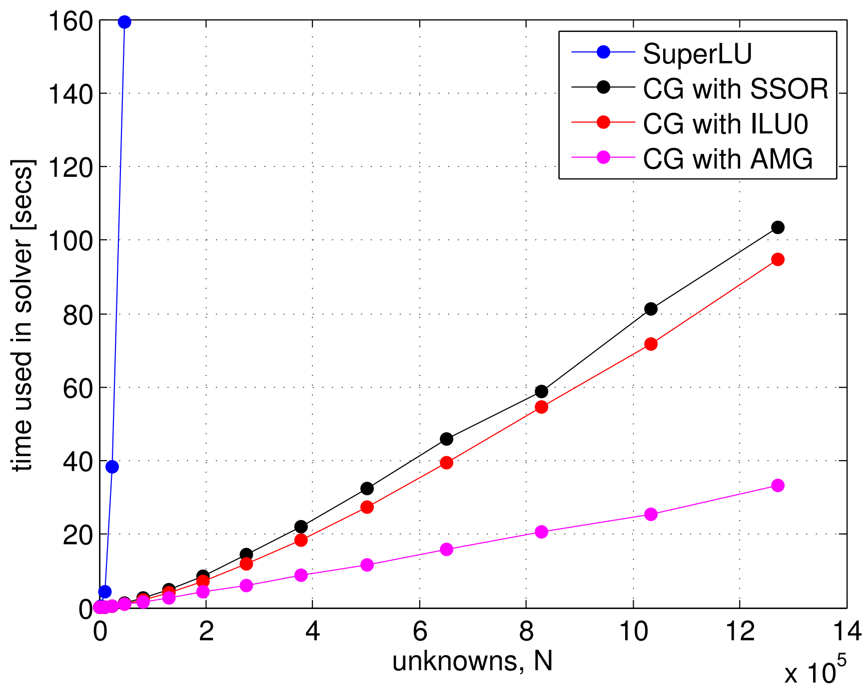


Figure 8.10: Time used by different linear solvers on the same problems. The conjugate gradient method with AMG as preconditioning is superior to the other alternatives investigated.

8.3.2 Analysis of Time Usage in the Cuboid Implementation

The time spent in the most time-consuming parts of the cuboid implementation is measured. The result of this is shown in Figure 8.11. Many parts of the implementation seem to scale almost perfectly with respect to the problem size. Perfect scaling means a linear relation between computation time and the number of elements.

The time used in the CG solver seem not to scale perfectly. The conjugate gradient method has the highest complexity, and also dominate the time usage in total. To achieve higher performance in total, this part should be looked into first.

The figure also shows that the calculation of the upscaled permeability is a very cheap operation, once the velocity solution is found.

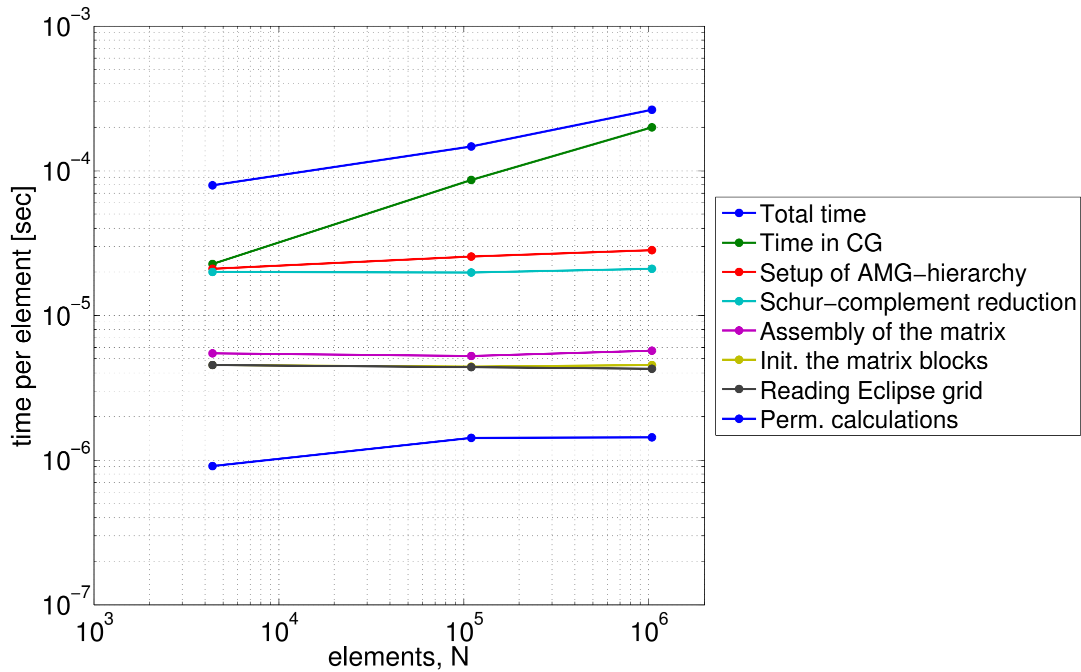


Figure 8.11: Time usage per element of different parts in the code. This figure indicates the scalability of the implementation.

8.4 Comments

The mixed and mixed hybrid finite element methods based on tetrahedron shaped elements give identical solutions, but the first method generate a linear system of equations which is too expensive to solve for large problems. The only difference between the two mixed hybrid methods is the shape of the elements used.

The method based on cuboid shaped elements is the most applicable for StatoilHydro since the Eclipse grid files used as input are uniform grids of cuboids. This grid type is a special case of the more general and widely used corner point grid. This is the reason why this implementation is analyzed thoroughly in the previous sections.

In the section where the efficiency is discussed, it is shown that the program developed in this thesis has lower performance than the program developed by Sintef. The reason for this is mainly the difference in the linear solver used. The solver used in the cuboid implementation has higher complexity than SAMG which was used in the other implementation. By tuning the parameters in the AMG preconditioner, it will probably be possible to achieve higher performance. Further development of the `gcc` compiler could also improve the performance by more optimizations aimed for the relative new template programming technique.

The DUNE framework is also in development, and have a great potential to be competitive with proprietary solvers. The users of DUNE can supply the community with experiences and results, and are important for the further development of the framework.

Chapter 9

Conclusion

There has been a rapid development in the DUNE framework within the last year. There are large changes in the core modules, and especially within the grid module. Several external modules based on the DUNE core modules are being developed. The result of this will reduce the complexity of implementing solvers such as the ones developed in this thesis.

The elliptic PDE discussed in chapter 3 was solved with three different implementations. The mixed method described in chapter 4 has the great disadvantage that the generated system of equations is not positive definite, and restricts the number of linear solvers which can be used. The poor scalability of this implementation, was the motivation for implementing the mixed hybrid method described in chapter 5. From the first working version of the implementation of the hybrid method to the final version, there has been a dramatic improvement in terms of performance. Since the software is GPL licensed, there are also possibilities for others to improve the implementation.

The existing program for permeability upscaling have been a great resource for the code development. The other implementation has worked as a measure in performance, and a goal to reach for.

The efficiency of the algebraic multigrid preconditioner in DUNE is superior compared to the other alternatives investigated. Compared with the proprietary SAMG solver, the performance is comparable for small problem sizes, while SAMG outperforms DUNE's AMG for larger problem sizes. It could be interesting to know if this is mostly due to more advanced optimizations in SAMG or if it is only related to the algorithmic difference of the two AMG solvers.

A great advantage with the implementation based on DUNE is that the memory usage is only 50% of Sintef's implementation for the largest problem investigated (approx. $3 \cdot 10^6$ unknowns). This can be of importance when the problem size is very large. One can imagine that Sintef's implementation could fail to run on a single machine due to the memory requirement, while the implementation based on DUNE runs, but uses longer time.

The open-source software for permeability upscaling based on DUNE has a potential to replace Sintef's implementation. The advantages are lower memory requirement and the GPL license. Currently, the greatest disadvantages are no support for more general Eclipse grids and slower performance. There is a great possibility for an increase in performance as a result of further development of the DUNE framework and the `g++`-compiler. The software developed in this thesis is open-source and makes it possible for everyone to improve and add functionality to the software.

9.1 Suggestions for Further Work

I suggest the following should be looked into:

- Support for more general Eclipse grids, and not just structured cuboid grids.
- Support for linear and periodic boundary conditions.
- Evaluate if the code should be based on the `dune-pdelab` module.

Bibliography

[1] GNU. General Public License, http://www.gnu.org/licenses/gpl.html.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A Generic Grid Interface for Parallel and Adaptive Scientific Computing. Part II: Implementation and Tests in DUNE. Submitted to Computing.

[3] The Visualization Toolkit. Homepage, http://www.vtk.org/.

[4] Wikipedia. Permeability, http://en.wikipedia.org/wiki/Permeability_(earth_sciences).

[5] Creative Commons. Attribution-Share Alike License, http://creativecommons.org/licenses/by-sa/2.5/.

[6] Dietrich Braess. *Finite elements. Theory, fast solvers, and applications in solid mechanics*. Cambridge University Press, 2007.

[7] SINTEF. Single-phase upscaling module for SBED/XMODEL. Technical report, SINTEF, 2006.

[8] Ricarcdo G. Durán. Mixed finite element methods. In *Lecture Notes in Mathematics, Mixed Finite Elements, Compatibility Conditions, and Applications*. Springer, 2008.

[9] C. Bahriawati and C. Carstensen. Three MATLAB implementations of the lowest-order Raviart-Thomas MFEM with a posteriori error control. *Computational Methods in Applied Mathematics*, 5:333–361, 2005.

[10] Franco Brezzi and Michael Frotin. *Mixed and Hybrid Finite Element Methods.* Springer Verlag, 1999.

[11] Peter Bastian. dune-pdelab - *Howto*.

[12] DUNE. Documentation, http://www.dune-project.org/doc/doxygen/html/.

[13] K. Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 2001.

[14] William K. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial. –2nd. edition.* SIAM, 2000.

[15] Adrian C. Muresan and Yvan Notay. Analysis of aggregation-based multigrid. *SIAM Journal on Scientific Computing*, 30(2):1082–1103, 2008.

Appendix A

A.1 Two Layered Eclipse Model

The two layered model shown in Figure 3.3 consisting of eight cubes with right angles in Eclipse grid format:

```
SPECGRID
2 2 2 1 F /

COORD
0.000 0.000 0.000 0.000 0.000 1.000
0.500 0.000 0.000 0.500 0.000 1.000
1.000 0.000 0.000 1.000 0.000 1.000
0.000 0.500 0.000 0.000 0.500 1.000
0.500 0.500 0.000 0.500 0.500 1.000
1.000 0.500 0.000 1.000 0.500 1.000
0.000 1.000 0.000 0.000 1.000 1.000
0.500 1.000 0.000 0.500 1.000 1.000
1.000 1.000 0.000 1.000 1.000 1.000
/

ZCORN
0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.000 0.000
0.000 0.000 0.000 0.000 0.500 0.500
0.500 0.500 0.500 0.500 0.500 0.500
0.500 0.500 0.500 0.500 0.500 0.500
0.500 0.500 0.500 0.500 0.500 0.500
0.500 0.500 0.500 0.500 0.500 0.500
0.500 0.500 0.500 0.500 0.500 0.500
1.000 1.000 1.000 1.000 1.000 1.000
1.000 1.000 1.000 1.000 1.000 1.000
1.000 1.000 1.000 1.000
/

ACTNUM
1  1  1  1  1  1  1  1 /

PERMX
0.01  0.01  0.01  0.01  4  4  4  4 /

PORO
0.2 0.2 0.2 0.2 0.2 0.2 0.2 0.2 /

SATNUM
1  1  1  1  2  2  2  2 /
```

A.2 Code Listing

This section is a code listing for the most important functions in the implementation of permeability upscaling on cuboid grids. For a simple code structure, see Figure 5.1.

The code is available at `http://folk.ntnu.no/arne/upscaling.tar.gz`. See section 6.3 for installation instructions.

A.2.1 Main Program

This is the file where the `main` function is located.

```
1 /***************************************************************************
2  * upscaling
3  *
4  ***************************************************************************
5  *
6  * Software for calculating upscaled permeability tensor using fixed
7  * boundary conditions on a homogeneous cuboid grid in the Eclipse grid
8  * format
9  *
10 * Copyright Arne Rekdal, 2009
11 *
12 ***************************************************************************
13 *
14 * upscaling is free software: you can redistribute it and/or modify
15 * it under the terms of the GNU General Public License as published by
16 * the Free Software Foundation, either version 3 of the License, or
17 * (at your option) any later version.
18 *
19 * upscaling is distributed in the hope that it will be useful,
20 * but WITHOUT ANY WARRANTY; without even the implied warranty of
21 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
22 * GNU General Public License for more details.
23 *
24 * You should have received a copy of the GNU General Public License
25 * along with upscaling.  If not, see <http://www.gnu.org/licenses/>.
26 *
27 ***************************************************************************/
28
29 #ifdef HAVE_CONFIG_H
30 # include "config.h"
31 #endif
32
33 #include <iostream>
34
35 //DUNE includes
36 #include<dune/common/mpihelper.hh>
37 #include<dune/common/exceptions.hh>
38 #include<dune/common/timer.hh>
39 #include<dune/grid/io/file/vtk/vtkwriter.hh>
40 #include<dune/grid/sgrid.hh>
41 #include<dune/istl/bvector.hh>
42 #include<dune/istl/io.hh>
43 #include<dune/istl/solvers.hh>
44 #include<dune/common/fvector.hh>
45 #include<dune/istl/paamg/amg.hh>
46 #include<dune/istl/paamg/pinfo.hh>
47
48 //Own header includes
49 #include"EclipseGridParser.h"
50 #include"eclipsegridinfo.h"
51 #include"rt0cube.h"
52
53 /***************************************************************************
54  * PROBLEM SPECIFICATION
55  ***************************************************************************/
```

```
56 template<class G, typename RT>
57 class FixedBC{
58   typedef typename G::template Codim<1>::Entity FaceEntity;
59   enum {dim=G::dimension};
60   typedef Dune::FieldVector<double,dim> CornerVector;
61   typedef Dune::FieldVector<double,dim> IntVector;
62 
63 public:
64   //Constructor for the fixed BC problem
65   FixedBC(int flowdimension_, const CornerVector& corner_,
66     const std::vector<double>& k_)
67     : flowdimension(flowdimension_), corner(corner_), k(k_) { }
68 
69   //Boolean function whether the face is on the Dirichlet boundary
70   bool onDirichletBoundary (const FaceEntity& e) const {
71     //Extracts the important component of the face's centroid
72     const double meanDim=
73       e.geometry().global(Dune::FieldVector<double,2>(0.25))[flowdimension];
74 
75     return ( fabs(meanDim-corner[flowdimension]) < 1e-10 ||
76       fabs(meanDim) < 1e-10 ) ;
77   }
78 
79   //Function returning the Dirichlet B.C. on the boundary face
80   RT dirichletValue(const FaceEntity& e){
81     const double meanDim=
82       e.geometry().global(Dune::FieldVector<double,2>(0.25))[flowdimension];
83 
84     //Return 1 if the face is located on the inhomogeneous Dirichlet boundary
85     return (RT) ( fabs(meanDim) < 1e-10 );
86   }
87 
88   //Function returning the inverse permeability in the element
89   RT kInverse(int index) const{ return 1.0/k[index]; }
90 
91   //Returns in which coordinate axis it is imposed a pressure drop
92   int flowDimension() const { return flowdimension; }
93 
94   //Returns the second corner of the grid (first is in the origin)
95   const CornerVector& getCorner() const { return corner; }
96 
97 private:
98   const int flowdimension;
99   const CornerVector& corner;
100   const std::vector<double> k;
101 };
102 
103 
104 
105 /*****************************************************************************
106  *
107  * MAIN METHOD
108  *
109  *****************************************************************************/
110 
111 int main(int argc, char** argv)
112 {
113   const int dim=3;
114 
115   //General type definitions
116   typedef double RT;
117   typedef Dune::SGrid<dim, dim> GridType;
118   typedef GridType::Codim<0>::LeafIterator ElementIterator;
119   typedef Dune::BlockVector< Dune::FieldVector<int,1> > IntVector;
120   typedef FixedBC<GridType, RT> Problem;
121   typedef RT0CubeAssembler<GridType, Problem, RT> Assembler;
122   typedef Assembler::BCRSMatrix Matrix;
123   typedef Assembler::Vector Vector;
```

```
124
125   //AMG specific type definitions
126   typedef Dune::MatrixAdapter<Matrix,Vector,Vector> Operator;
127   typedef Dune::Amg::CoarsenCriterion
128     <Dune::Amg::UnSymmetricCriterion<Matrix,Dune::Amg::FirstDiagonal> > Criterion;
129   typedef Dune::SeqILU0<Matrix,Vector,Vector> Smoother;
130   typedef Dune::Amg::SmootherTraits<Smoother>::Arguments SmootherArgs;
131   typedef Dune::Amg::AMG<Operator,Vector,Smoother> AMG;
132
133   try{
134     //Input check
135     bool writeOutput=false;
136     double relax=1;
137     if (argc<2){
138       std::cout << "No Eclipse grid-file specified. Aborting." << std::endl;
139       return(1);
140     }
141     if(argc>2) writeOutput=atoi(argv[2]);
142     if(argc>3) relax=atof(argv[3]);
143
144     //Parsing Eclipse file
145     std::cout << "Running upscaling on <" << argv[1] << ">." << std::endl;
146     char* eclipsefile=argv[1];
147     EclipseGridParser gp(eclipsefile);
148     EclipseGridInfo eclgridinfo(gp);
149
150     //Grid creation
151     const Dune::FieldVector<GridType::ctype, dim> L(0);
152     const Dune::FieldVector<GridType::ctype, dim>& H=eclgridinfo.getCorner();
153     const Dune::FieldVector<int, dim>& N=eclgridinfo.getGridSize();
154     Dune::FieldVector<bool, dim> periodic(false);
155     GridType grid(N,L,H);
156
157     //Solving three flow problems, one per coordinate axis
158     for( int flowdimension=0; flowdimension<dim; flowdimension++){
159
160       //Creating the problem and assembles the system of equations
161       Problem problem(flowdimension, H, gp.getFloatingPointValue("PERMX"));
162       Assembler assembler(grid, problem);
163       assembler.assemble();
164       assembler.schurReduction();
165
166       //Extracts the system of equations
167       Matrix& S=assembler.getMatrixS();
168       Vector& r=assembler.getRHSVector();
169
170       //Setting up the AMG preconditioner
171       Operator op(S);
172       Vector pi(r);
173       SmootherArgs smootherArgs;
174       smootherArgs.relaxationFactor=relax;
175       Criterion criterion;
176
177       //Creating the AMG preconditioner
178       AMG prec(op, criterion, smootherArgs, 1,1, 1, false);
179
180       //Convergence criterion of CG
181       double reduction=1e-8;
182       int maxit=S.N();
183       int verbose=1;
184
185       //Solve the system of equations
186       Dune::CGSolver<Vector> solver(op,prec,reduction,maxit,verbose);
187       Dune::InverseOperatorResult res;
188       solver.apply(pi,r,res);
189
190       //Back-substitution
191       Vector p(grid.size(0));
```

```
192       Vector v(6*grid.size(0));
193       assembler.solvePressure(pi,p);
194       assembler.solveVelocity(pi,p,v);
195
196       //Export the solution and perm field along the z-coordinate axis if requested
197       if (writeOutput && flowdimension==dim-1 ) {
198         Dune::VTKWriter<GridType::LeafGridView> vtkwriter(grid.leafView());
199         vtkwriter.addCellData(p, "pressure");
200         vtkwriter.addCellData(gp.getFloatingPointValue("PERMX"), "permeability");
201         vtkwriter.addVertexData(new RT0CubeVelocityFunction<GridType, Vector>(grid,
                v));
202         vtkwriter.write("upscaling");
203       }
204
205       //Calculates the upscaled permeability
206       const double perm=calculatePerm(grid, v, problem);
207       printf("  K(%i, %i)=%f\n", flowdimension, flowdimension, perm);
208     }
209     return 0;
210   }
211   catch (Dune::Exception &e){
212     std::cerr << "Dune reported error: " << e << std::endl;
213   }
214   catch (...){
215     std::cerr << "Unknown exception thrown!" << std::endl;
216   }
217 }
```

A.2.2 RTOCubeLocal Class

Code listing for the `RT0CubeLocal` class. The functionality of this class is discussed in section 5.3.1. In the listing below, `D` denotes the Π-block described in chapter 5.

```
1 /**************************************************************************
2  * Local assembler
3  **************************************************************************/
4 template<class GV,class RT>
5 class RT0CubeLocal{
6   //Ectracts the type information from the grid
7   typedef typename GV::Grid::ctype DT;
8   typedef typename GV::template Codim<0>::Entity ElementEntity;
9   typedef typename GV::template Codim<1>::Entity FaceEntity;
10  typedef typename GV::IntersectionIterator IntersectionIterator;
11
12 public:
13   // Types for matrics, vectors and boundary conditions
14   enum { dim=GV::dimension };
15   const static int dof=6;
16   typedef Dune::FieldMatrix<RT,dof,dof> Matrix; // Diagonal matrix for B_local
17   typedef Dune::FieldVector<RT,dof> Vector; // Vector for holding C_local
18   typedef Dune::BlockVector<Vector> BlockVector; // Vector of Vectors, for D_local
19   typedef Dune::FieldVector<DT,dim> CVector; // Coordinate Vector
20
21   // Constructor
22   RT0CubeLocal() : D_(dof){ }
23
24   ~RT0CubeLocal() {}
25
26   // Contribution from one element
27   void assemble (const ElementEntity& e, RT Kinv){
28     RT volume=e.geometry().volume();
29     const CVector midPoint=e.geometry().global(CVector(0.5));
30     const CVector lowerCorner=e.geometry().global(CVector(0));
31     const CVector upperCorner=e.geometry().global(CVector(1));
32     const CVector h=upperCorner-lowerCorner;
33
34     //Finding the quadrature rule for this element type
```

```
35     const int p=3;
36     Dune::GeometryType gt=e.type();
37     const Dune::QuadratureRule<DT, dim>& rule=Dune::QuadratureRules<DT,dim>::rule(
           gt,p);
38
39     //Fills the local B stiffness matrix by applying a quadrature
40     B_=0;
41     for(typename Dune::QuadratureRule<DT, dim>::const_iterator i=rule.begin(); i!=
           rule.end(); ++i){
42       CVector qp1=(e.geometry().global(i->position()));
43       for (int j=0; j<dim; j++) qp1[j]=(qp1[j]-lowerCorner[j])/h[j];
44       CVector qp2(qp1); qp1*=-1; qp1+=1;
45
46       double b11, b22, b33, b44, b55, b66, b12, b34, b56;
47       b11=b22=b33=b44=b55=b66=b12=b34=b56=0;
48
49       //B_11, B_33, B_55
50       b11+=qp1[0]*qp1[0];
51       b33+=qp1[1]*qp1[1];
52       b55+=qp1[2]*qp1[2];
53
54       //B_22, B44, B_66
55       b22+=qp2[0]*qp2[0];
56       b44+=qp2[1]*qp2[1];
57       b66+=qp2[2]*qp2[2];
58
59       //B_12, B_34, B_56
60       b12+=qp1[0]*qp2[0];
61       b34+=qp1[1]*qp2[1];
62       b56+=qp1[2]*qp2[2];
63
64       const double weight=i->weight();
65       const double detjac=e.geometry().integrationElement(i->position());
66       const double factor=weight*detjac;
67
68       B_[0][0]+=b11*factor; B_[1][1]+=b22*factor;
69       B_[2][2]+=b33*factor; B_[3][3]+=b44*factor;
70       B_[4][4]+=b55*factor; B_[5][5]+=b66*factor;
71
72       B_[0][1]+=b12*factor;
73       B_[2][3]+=b34*factor;
74       B_[4][5]+=b56*factor;
75     }
76     B_[1][0]=B_[0][1];
77     B_[3][2]=B_[2][3];
78     B_[5][4]=B_[4][5];
79     B_*=Kinv;
80
81     //Fills the C vector
82     C_[0]=-volume/h[0]; C_[1]=volume/h[0];
83     C_[2]=-volume/h[1]; C_[3]=volume/h[1];
84     C_[4]=-volume/h[2]; C_[5]=volume/h[2];
85
86     //Fills the local D vectors, each corresponding to a face of the element
87     int alpha=0;
88     for(IntersectionIterator is=e.ileafbegin(); is!=e.ileafend(); ++is, alpha++){
89       CVector normal=is->unitOuterNormal(Dune::FieldVector<RT,2>(0.5));
90       RT facearea =is->intersectionGlobal().volume();
91       Vector Dalpha(0);
92       const int sign=((alpha+1)%2)?1:-1;
93       Dalpha[alpha]=-sign*facearea;
94       D_[alpha]=Dalpha;
95     }
96   }
97
98   RT B(int alpha, int beta){ return B_[alpha][beta]; }
99
100  RT C(int alpha){ return C_[alpha]; }
```

```
101
102   Vector D(int alpha){ return D_[alpha]; }
103
104 private:
105   Matrix B_; // Corresponds to B_lcoal
106   Vector C_;  // Corresponds to C_local
107   BlockVector D_; // Corresponds to D_local
108 };
```

A.2.3 RTOCubeAssembler Class

Code listing for the `RT0CubeAssembler` class. This class creates the system of equations, and tranforms the system into a symmetric positive definite system with the function `schurReduction`. Once the solution of this system is found, e.g. by an iterative method, the solution can be transformed back to the original variables with the functions `solvePressure` and `solveVelocity`.

In the listing below, `D` denotes the Π-block described in chapter 5, and the transformed matrix D in (5.16) is denoted `E`.

```
1  /*****************************************************************************
2   * Assembler of the system of equations
3   *****************************************************************************/
4  template<class G, class P, typename RT>
5  class RT0CubeAssembler{
6    typedef typename G::template Codim<0>::LeafIterator ElementIterator;
7    typedef typename G::template Codim<1>::LeafIterator FaceIterator;
8    typedef typename G::template Codim<0>::Entity ElementEntity;
9    typedef typename G::template Codim<1>::Entity FaceEntity;
10   typedef typename G::LeafGridView::IntersectionIterator IntersectionIterator;
11   enum {dim=G::dimension };
12
13  public:
14   typedef typename Dune::template BCRSMatrix<Dune::FieldMatrix<RT,1,1> > BCRSMatrix
       ;
15   typedef typename Dune::template BlockVector<Dune::FieldVector<RT,1> > Vector;
16   typedef typename Dune::FieldVector<RT,6> FixedVector;
17   typedef typename Dune::template BlockVector<Dune::FieldVector<int,1> > IntVector;
18
19   // Constructor
20  RT0CubeAssembler(const G& g, const P& p) : grid(g), problem(p), elementfaces(grid.
       size(0)), f2dof(grid.size(1)) {
21     lagrangemult=0;
22     init();
23   }
24
25   // Destructor
26   ~RT0CubeAssembler() {
27
28   }
29
30   // Init function: Setting up the structure of the global blocks B, C, D.
31   void init() {
32     // Finds the number of faces that are unknown Lagrange multipliers
33     for (FaceIterator it=grid.template leafbegin<1>(); it!= grid.template leafend
         <1>(); ++it){
34       int faceIndex=grid.leafIndexSet().index(*it);
35       if (! problem.onDirichletBoundary(*it)){
36         f2dof[faceIndex]=lagrangemult++;
37       } else {
38         f2dof[faceIndex]=-1;
39       }
40     }
41
42     //=============================
```

```
43     // Sets up the global B block
44     //=============================
45     int systemsize=6*grid.size(0);
46     B.setBuildMode(BCRSMatrix::row_wise);
47     B.setSize(systemsize,systemsize,2*systemsize);
48     bd.resize(systemsize);
49     bd=0;
50
51     typedef typename BCRSMatrix::CreateIterator Iter;
52     for(Iter row=B.createbegin(); row!=B.createend(); ++row){
53       // Add nonzeros on the diagonal
54       row.insert(row.index());
55
56       // Above/below the diagonal?
57       const int sign=((row.index()+1)%2)?1:-1;
58       row.insert(row.index()+sign);
59     }
60     B=0;
61
62
63     //=============================
64     //Sets up the global C block
65     //=============================
66     C.setBuildMode(BCRSMatrix::row_wise);
67     C.setSize(grid.size(0),systemsize,6*grid.size(0));
68
69     for(Iter row=C.createbegin(); row!=C.createend(); ++row){
70       // Add nonzeros on the diagonal
71       int index=6*row.index();
72       for(int alpha=0; alpha<6; alpha++)
73         row.insert(index+alpha);
74     }
75     C=0;
76
77     //=============================
78     //Sets up the global D block
79     //=============================
80     D.setBuildMode(BCRSMatrix::random);
81     D.setSize(lagrangemult, systemsize);
82     face2elements.resize(lagrangemult); face2elements=-1;
83     for(int i=0; i<lagrangemult; i++){ D.setrowsize(i,0); }
84
85     // STEP 1: Determine rowsizes.
86     for(ElementIterator it=grid.template leafbegin<0>(); it!= grid.template leafend
           <0>(); ++it){
87       int elementIndex=grid.leafIndexSet().index(*it);
88
89       for (int alpha=0; alpha<6; alpha++){
90         int indexAlpha=grid.leafIndexSet().template subIndex<1>(*it,alpha);
91         int dofIndex=f2dof[indexAlpha];
92
93         // Checks if the face is a non-Dirichlet face
94         if ( dofIndex >=0 ){
95           if (face2elements[dofIndex][0]==-1)
96             face2elements[dofIndex][0]=elementIndex;
97           else
98             face2elements[dofIndex][1]=elementIndex;
99
100          D.incrementrowsize(f2dof[indexAlpha],6);
101        }
102      }
103    }
104    D.endrowsizes();
105
106    // STEP 2: Determine indices
107    for(ElementIterator it=grid.template leafbegin<0>(); it!= grid.template leafend
           <0>(); ++it){
108      int elementIndex=grid.leafIndexSet().index(*it);
```

```
109       for (int alpha=0; alpha<6; alpha++){
110         int indexAlpha=grid.leafIndexSet().template subIndex<1>(*it,alpha);
111 
112         // faceAlpha is a DOF
113         if (f2dof[indexAlpha] >=0) {
114           // Adds the index corresponding to the C block
115           for (int i=0; i<6; i++)
116             D.addindex(f2dof[indexAlpha], 6*elementIndex+i);
117         }
118       }
119     }
120     D.endindices();
121     D=0;
122   }
123 
124   // Function for filling the system of equations once the blocks are set up
125   void assemble(){
126     for(ElementIterator it=grid.template leafbegin<0>(); it!= grid.template leafend
          <0>(); ++it){
127       int elementIndex=grid.leafIndexSet().index(*it);
128 
129       //Assemble the elementstiffness matrix
130       double kinv=problem.kInverse(elementIndex);
131       rt0local.assemble(*it, kinv);
132 
133       //Assemble the B, C and D block and the rhs vector bd
134       for (int alpha=0; alpha<6; alpha++){
135         const int faceindex=6*elementIndex+alpha;
136         B[faceindex][faceindex]=rt0local.B(alpha, alpha);
137         const int sign=((alpha+1)%2)?1:-1;
138         B[faceindex][faceindex+sign]=rt0local.B(alpha,alpha+sign);
139 
140         int lagrangeindex=f2dof[grid.leafIndexSet().template subIndex<1>(*it,alpha)
              ];
141         elementfaces[elementIndex][alpha]=lagrangeindex;
142 
143         if(lagrangeindex>=0){
144           //Face is NOT a Dirichlet Face
145           for(int beta=0; beta<6; beta++)
146             D[lagrangeindex][6*elementIndex+beta]=rt0local.D(alpha)[beta];
147         } else {
148           //Face is a Dirichlet Face
149           RT p=problem.dirichletValue(*(it->template entity<1>(alpha)) );
150           for(int beta=0; beta<6; beta++)
151             bd[6*elementIndex+beta]=-p*rt0local.D(alpha)[beta];
152         }
153         C[elementIndex][faceindex]=rt0local.C(alpha);
154       }
155     }
156     return;
157   }
158 
159   //Function for transforming the system of equations
160   void schurReduction(){
161     typedef typename BCRSMatrix::RowIterator RowIter;
162 
163     //==============================
164     //Inverts B
165     //==============================
166     for(RowIter row=B.begin(); row!=B.end(); ++row){
167       const int i=row.index();
168       const double determinant=B[i][i]*B[i+1][i+1]-B[i][i+1]*B[i+1][i];
169 
170       const double temp=B[i+1][i+1];
171       B[i+1][i+1]=B[i][i]/determinant;
172       B[i][i]=temp/determinant;
173       B[i][i+1]/=-determinant;
174       B[i+1][i]/=-determinant;
```

```
175
176       ++row;
177     }
178
179     //==============================
180     //Calculate diagonal E matrix
181     //==============================
182     E.resize(C.N()); E=0;
183     for(unsigned int i=0; i<C.N(); i++){
184       for(int alpha=0; alpha<6; alpha++){
185         const signed int sign=2*((alpha+1)%2)-1;
186         const int k1=6*i+alpha;
187         const int k2=6*i+alpha+sign;
188
189         E[i]+=C[i][k1]*( B[k1][k1]*C[i][k1] + B[k1][k2]*C[i][k2] );
190       }
191     }
192
193     //==============================
194     //Sets up the global F block
195     //==============================
196     typedef typename BCRSMatrix::RowIterator RowIter;
197     typedef typename BCRSMatrix::ColIterator ColIter;
198     typedef typename BCRSMatrix::ConstRowIterator ConstRowIter;
199     typedef typename BCRSMatrix::ConstColIterator ConstColIter;
200
201     typedef typename BCRSMatrix::row_type Row;
202
203     F.setBuildMode( BCRSMatrix::random);
204     F.setSize(D.M(), C.M());
205
206     for(ConstRowIter row=D.begin(); row!=D.end(); ++row){
207       for(ConstColIter col=row->begin(); col!=row->end(); ++col){
208         int j=col.index();
209         if ((j+1)%6 ==0)
210           F.incrementrowsize(row.index());
211       }
212     }
213     F.endrowsizes();
214     for(ConstRowIter row=D.begin(); row!=D.end(); ++row){
215       for(ConstColIter col=row->begin(); col!=row->end(); ++col){
216         int j=col.index();
217         if ((j+1)%6 ==0){
218           int colindex=(int)floor(j/6);
219           F.addindex(row.index(),colindex);
220         }
221       }
222     }
223     F.endindices();
224     F=0;
225
226     for(ConstRowIter row=D.begin(); row!=D.end(); ++row){
227       int i=row.index();
228       for(ConstColIter col=row->begin(); col!=row->end(); ++col){
229         int j=col.index();
230         int colindex=(int)floor(j/6);
231         const int sign=2*((j+1)%2)-1;
232         const double term1=B[j][j]*C[colindex][j];
233         const double term2=B[j][j+sign]*C[colindex][j+sign];
234         F[i][colindex]+=D[i][j]*(term1+term2);
235       }
236     }
237
238     //==============================
239     //Create S matrix
240     //==============================
241     //STEP1: Determine row-sizes
242     S.setBuildMode(BCRSMatrix::row_wise);
```

```
243     S.setSize(D.N(), D.N());
244
245     typedef typename BCRSMatrix::CreateIterator Iter;
246
247     for(Iter row=S.createbegin(); row!=S.createend(); ++row){
248       const int elementP=face2elements[row.index()][0];
249       const int elementM=face2elements[row.index()][1];
250
251       for (int alpha=0; alpha<6; alpha++){
252         if (elementP >=0)
253           if(elementfaces[elementP][alpha]>=0)
254             row.insert( elementfaces[elementP][alpha] );
255         if (elementM >=0)
256           if(elementfaces[elementM][alpha]>=0)
257             row.insert( elementfaces[elementM][alpha] );
258       }
259     }
260     S=0;
261
262     //STEP3: Fill the matrix
263     for(RowIter row=S.begin(); row!=S.end(); ++row){
264       const int i=row.index();
265       for(ColIter col=row->begin(); col!=row->end(); ++col){
266         const int j=col.index();
267
268         //The first term: D*Binv*D'
269         ConstColIter col_i=D[i].begin(); ConstColIter col_j=D[j].begin();
270         while (col_i != D[i].end() && col_j != D[j].end() ) {
271           const int m=col_i.index();
272           const int n=col_j.index();
273           const int sign=( (m+1)%2 ) ? 1 : -1;
274
275           if( m==n) {
276             *col+=(*col_i * *col_j)*(B[m][m]);
277             *col+=(*col_i * D[j][m+sign])*B[m][m+sign];
278             ++col_j;
279           } else if (m<n) ++col_i;
280           else ++col_j;
281         }
282
283         //The second term: -F*Einv*F'
284         const Row& Frow_i=F[i];
285         const Row& Frow_j=F[j];
286         col_i=Frow_i.begin(); col_j=Frow_j.begin();
287         while (col_i != Frow_i.end() && col_j != Frow_j.end() ) {
288           const int m=col_i.index();
289           const int n=col_j.index();
290
291           if (m == n) {
292             *col-=(*col_i * *col_j)/E[m];
293              ++col_i;
294              ++col_j;
295           } else if (m < n) ++col_i;
296           else ++col_j;
297         }
298       }
299     }
300
301     //==============================
302     //Filling RHS-Vector
303     //==============================
304     Vector rtemp(B.N());
305     typedef typename Vector::Iterator VecIt;
306
307     //Binv*bd
308     B.mv(bd,rtemp);
309
310     //(D-F*Einv*C)*Binv*bd
```

```
311     r.resize(D.N());
312     D.mv(rtemp, r); // r=D*Binv*bd2
313
314     BCRSMatrix EinvC(C);
315     for(RowIter row_i=EinvC.begin(); row_i!=EinvC.end(); ++row_i){
316       const int i=row_i.index();
317       const RT Eii=E[i];
318
319       for(ColIter col_j=row_i->begin(); col_j!=row_i->end(); ++col_j)
320         *col_j/=Eii;
321     }
322     schurtemp.resize(C.N());
323     EinvC.mv(rtemp,schurtemp); //schurtemp=Einv*C*Binv*bd
324     F.mmv(schurtemp, r); // r-=F*schurtemp
325   }
326
327   // Method for calculating the pressure in each element,
328   // when the Lagrange multiplicators are found
329   void solvePressure(const Vector& pi, Vector& p){
330     p=0;
331     F.umtv(pi,p); // Calculate p+=F^T*pi
332
333     //Calculate p=Einv*p
334     typedef typename Vector::Iterator VecIt;
335     for(VecIt elem=p.begin(); elem!=p.end(); ++elem){
336       const int i=elem.index();
337       const RT Eii=E[i];
338       *elem/=Eii
339     }
340     p-=schurtemp;
341   }
342
343   void solveVelocity(const Vector& pi, const Vector& p, Vector& v){
344     //v=binv*(bd+C^T*p-D^T*pi)
345     Vector temp(bd);
346     C.umtv(p, temp);
347     D.mmtv(pi, temp);
348
349     B.mv(temp,v);
350   }
351
352   // Getters for the matrix system
353   const BCRSMatrix& getMatrixS() const{ return S; }
354
355   BCRSMatrix& getMatrixS(){ return S; }
356
357   Vector& getRHSVector(){ return r; }
358
359   IntVector& getFaceMap(){ return f2dof; }
360
361  private:
362   const G& grid;
363   P problem;
364   Dune::BlockVector< Dune::FieldVector<int, 6> > elementfaces;
365   Dune::BlockVector< Dune::FieldVector<int, 2> > face2elements;
366   int flowDimension, lagrangemult;
367   IntVector f2dof;
368   BCRSMatrix B, C, D, F, S;
369   Vector E, bd, bn, r, schurtemp;
370   RT0CubeLocal<typename G::LeafGridView, RT> rt0local;
371 };
```

A.2.4 Utility Function for Calculation of Upscaled Permeability

Code listing for the function which calculates the upscaled permeability.

```
1 /***************************************************************************
2  * Function for calculating the upscaled permeability
3  **************************************************************************/
4 template<class G, class V, class P>
5 double calculatePerm(const G& grid, const V& vel, const P& problem){
6   enum {dim=G::dimension};
7   typedef typename G::template Codim<0>::Entity Entity;
8   typedef typename G::template Codim<1>::Entity FaceEntity;
9   typedef typename G::template Codim<0>::LeafIterator ElementIterator;
10
11   typedef typename G::ctype ctype;
12   typedef Dune::FieldVector<ctype,dim> CVector; // Coordinate Vector
13
14   double perm=0;
15   double integrand0=0, integrand1=0;
16   double area0=0, area1=0;
17   const int flowdimension=problem.flowDimension();
18
19   for(ElementIterator it=grid.template leafbegin<0>(); it!=grid.template leafend
        <0>(); ++it){
20     const int elementIndex=grid.leafIndexSet().index(*it);
21     for(int alpha=0; alpha<6; alpha++){
22
23       // Checks if this face is on the Dirichlet boundary
24       // => This is included in the average flux
25       if ( problem.onDirichletBoundary(*(it->template entity<1>(alpha))) ) {
26         const double facearea=it->template entity<1>(alpha)->geometry().volume();
27         const double velocity=vel[6*elementIndex+alpha];
28         const double meanDim=it->template entity<1>(alpha)->geometry().global(Dune
              ::FieldVector<double,2>(0.5))[flowdimension];
29         if ( fabs(meanDim) < 1e-10 ) { // Located on the "bottom"
30             integrand0+=facearea*velocity;
31             area0+=facearea;
32         } else {
33             integrand1-=facearea*velocity;
34             area1+=facearea;
35         }
36       }
37     }
38   }
39   perm=-problem.getCorner()[flowdimension]*( integrand1/(2*area1)-integrand0/(2*
        area0) );
40   return perm;
41 };
```