NTNU – Trondheim
Norwegian University of
Science and Technology

Multiscale Finite Volume Methods

Extension to Unstructured Grids with
Applications in Reservoir Simulation

Olav Møyner

Master of Science in Physics and Mathematics
Submission date: June 2012
Supervisor: Helge Holden, MATH
Co-supervisor: Knut-Andreas Lie, SINTEF IKT

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

In reservoir simulations, one of the biggest challenges is solving large models with complex geological properties. Because reservoirs can be several kilometers long, and still be geologically inhomogeneous over centimeter scales, the computational power required to solve a full set of mass balance equations can be immense. Several methods for overcoming this challenge has been proposed, including various upscaling and multiscale methods.

One of these methods is the Multiscale Finite Volume (MsFV) method, which aims to create a set of basis functions for the pressure which can be computed in parallel and reused for different boundary conditions. This thesis aims to give a thorough study of the MsFV-method itself, before extending it to three dimensional, unstructured grids. An implementation was done as a module for the MATLAB Reservoir Simulation Toolbox developed by SINTEF Applied Mathematics. A new variant of the method designed to overcome some of the computational challenges arising from an extension to 3D was also formulated.

The implementation was then applied to both synthetic and realistic grids and permeabilities, and compared against a full two point flux approximation (TPFA) solver.

Norsk sammendrag

En av de største utfordringene innen reservoarsimulering i dag oppstår ved løsning av store modeller med komplekse geologiske egenskaper. Siden reservoarer kan være flere kilometer lange og allikevel være svært inhomogene fra centimeter til centimeter kan det være svært beregningskrevende å løse et fullt sett med massebevaringslikninger for et slikt system. Flere metoder for å håndtere slike modeller har blitt utviklet, deriblant forskjellige oppskalering- og flerskalametoder.

En av disse metodene er den flerskala endelige volummetoden (MsFV) som konstruerer et sett basisfunksjoner for systemtrykket som kan beregnes parallelt og gjenbrukes for forskjellige grensebetingelser. Denne oppgaven forsøker først å gi en grundig gjennomgang av MsFV-metoden, som deretter utvides til tredimensjonale, ustrukturerte gitter. En implementasjon ble gjort som en modul til SINTEF Anvendt matematikk sin MATLAB Reservoir Simulation Toolbox. Oppgaven inneholder også en formulering av en ny metode som er laget med hensyn på å løse problemer som oppstår fra generaliseringen til 3D.

Implementasjonene ble deretter testet på både syntesiske og realistiske gitter og permeabiliteter, og løsningene ble sammenlignet mot en full topunkts-løser.

Preface

This thesis is the last part of my master's degree in Industrial Mathematics at the Norwegian University of Science and Technology (NTNU). The work was performed from the 16th of January to the 1th of June 2012. The theoretical foundation for much of the thesis comes from my project assignment done in the fall semester of 2011 and some figures and sections[1] are based on this work.

The thesis was written with help from SINTEF Applied Mathematics and I would like to thank Knut-Andreas Lie for giving me the opportunity to write about an interesting subject with real life applications, as well as guidance and corrections during the last year. Helge Holden also deserves thanks, for being my local supervisor at NTNU and giving me feedback on my drafts. I would also like to thank Bård Skaflestad at SINTEF for help with understanding the method and the intricacies of MRST.

Last but not least, I would like to thank my friends and family, who have distracted me from the thesis when necessary so I could have a functional social life during this last semester. Bob Dylan should also be credited, since he accompanied me on late night coding sessions when no-one else could.

Olav Møyner
Trondheim,
June 1, 2012

[1] Chapter 1 and the first parts of Chapter 2 and 6. Otherwise where noted.

Contents

Chapter 1

Theoretical background

1.1 Problem statement

The model problem we will study has its origins in the study of fluid mechanics for reservoir flow. Our primary example will be incompressible single phase flow in and around porous media, modelled by the continuity equation

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla \cdot (\rho v) = q, \tag{1.1}$$

where ρ is the fluid density, ϕ a porosity distribution, v the fluid velocity and q any source and sink terms. We will study the incompressible flow problem where the fluid density is a constant so that time derivative vanishes and leaves us with the simple equation

$$\nabla \cdot (\rho v) = q \tag{1.2}$$

$$\nabla v = \frac{q}{\rho} \tag{1.3}$$

which states that flux out of some closed surface is equal to the source and sink terms on the inside. For slow moving fluids, like on the onside of an reservoir, we can apply Darcy's law:

$$v = -\frac{K}{\mu}(\nabla p + \rho g \nabla z), \tag{1.4}$$

$$= -\lambda(\nabla p + \rho G) \tag{1.5}$$

which models filtration through media like sand, porous rock and so on. Essentially, the term inside the parenthesis describe the forces applied in the fluid: There is a pressure term ∇p which models the force arising from a difference in pressure over some distance, as well as a term $\rho g \nabla z = \rho G$ which describes the fluid column pressing down giving rise to a difference in pressure. The pressure is then multiplied with $\lambda = \frac{K}{\mu}$ which is the permeability tensor divided by the fluid viscosity, which implies that higher viscosity leads to slower flow in a linear way.

By inserting the expression for fluid velocity from Darcy's law into (1.2) we get

$$\nabla \cdot v = \nabla \cdot [-\lambda (\nabla p - \rho G)] = \frac{q}{\rho} \tag{1.6}$$

For the purpose of testing the multiscale finite volume (MsFV) method we will consider the scenario where gravity is negligible (the gravity contribution is a separate term and can easily be integrated into a more generic function on the right hand side along with the source and sink terms):

$$-\nabla \cdot \lambda \nabla p = \frac{q}{\rho} \tag{1.7}$$

1.2 Finite volume methods

Finite volume methods for solutions of partial differential equations are interesting when it comes to solving flow problems because they are conservative. This means that any solutions will ensure that the property we are solving the equations for, typically some fluid, will remain constant when considering a problem without source terms. This is not to say that other approaches like finite difference schemes or finite element methods always produce unwanted quantities when solving fluid problems, rather that they cannot be *guaranteed* not to do so. The different methods can sometimes lead to both the same results and the same sets of linear equations, leaving the choice between different methods more of a choice between different perspectives on the same problem instead of entirely different approaches.

While other methods for solving partial differential equations like finite difference methods (FDM) or finite element methods (FEM) originate from mathematical considerations, finite volume methods (FVM) have their origins in a more physical understanding of flow problems. While for example FDM consists of Taylor expanding different terms of the differential equation around

grid points to achieve a linear equation set, under the assumption of the solution is sufficiently smooth without any obvious physical interpretation, the FVM approach is to consider the problem divided into cells and then applying conservation laws over the cell edges to obtain a linear equation set.

The multiscale solver bases itself on a discretization of (1.7) from an exisiting method, and for this purpose we will need the two point flux finite volume approximation (TPFA), which is the most common method used in reservoir simulators today. A 2D FVM formulation with equidistant grid size in both x and y directions $h = h_x = h_y$ will be used as an example to simplify the notation, and the calculations done for unstructured grids in 3D is a straight forward extension.

We will quickly restate the notation for such problems, which should be familiar to anyone who has previous experience with numerical analysis:

$$h = \frac{x_n - x_0}{n}, \tag{1.8}$$

$$x_i = x_0 + ih, \tag{1.9}$$

$$y_i = y_0 + ih \tag{1.10}$$

$$\tag{1.11}$$

where $n + 1$ is the number of grid points in both directions.

We note that (1.4), when omitting the gravitational pull, states the fluid velocity as

$$v = -\lambda \nabla p. \tag{1.12}$$

If we integrate (1.7) over some square cell Ω in 2D we get

$$-\int_\Omega \nabla \cdot \lambda \nabla p = \int_\Omega \frac{q}{\rho}. \tag{1.13}$$

Since the boundary is piecewise smooth and the domain is compact we can use the divergence theorem

$$\int_\Omega (\nabla \cdot \mathbf{u}) dA = \int_{\partial\Omega} (\mathbf{u} \cdot \mathbf{n}) dS \tag{1.14}$$

$$\tag{1.15}$$

to reformulate the problem as

$$-\int_{\partial\Omega} (\lambda \nabla \mathbf{p} \cdot \mathbf{n}) dS = \int_\Omega \frac{q}{\rho} dA. \tag{1.16}$$

By using the notation

$$q^x_{i+1/2,j} = -\int_{\partial\Omega_i} (\lambda \nabla \mathbf{p} \cdot \mathbf{n_i}) dS \tag{1.17}$$

where q_i is the flux out of the edge corresponding to cell (i,j) in x-direction on the right cell wall (denoted by the index $i + 1/2$), we can describe the total flux out of cell (i,j) as

$$q_{up} + q_{down} + q_{left} + q_{right} = \text{sources inside} \tag{1.18}$$

$$q^y_{i,j+1/2} + q^y_{i,j-1/2} + q^x_{i-1/2,j} + q^x_{i+1/2,j} = \int_\Omega \frac{q}{\rho} dA. \tag{1.19}$$

$$\tag{1.20}$$

Since we want equations for $u_i j$, we need to relate the cell edge fluxes to the pressure. We begin by expressing the flux in one direction as

$$\frac{\partial p_x}{\partial x} = -\frac{q_x}{\lambda^x}. \tag{1.21}$$

We integrate from cell i to $i + 1$ and get

$$\int_{x_i}^{x_{i+1}} \frac{\partial p_x}{\partial x} dx = -\int_{x_i}^{x_{i+1}} \frac{q_x}{\lambda^x} dx, \tag{1.22}$$

$$p_{i+1,j} - p_{i,j} = -\int_{x_i}^{x_{i+1}} \frac{q_x}{\lambda^x} dx. \tag{1.23}$$

$$\tag{1.24}$$

We approximate the integral on the right hand side by setting the flux to a constant value between the cell centers since we seek cell wise constant pressure. For λ^x we can simply solve the integral analytically when keeping in mind that λ^x is also defined as cell wise constant. Since h is a constant in our model, one half of both the cells will be included in the integral.

$$p_{i+1,j} - p_{i,j} = -q_{i+1/2,j} \int_{x_i}^{x_{i+1}} \frac{1}{\lambda^x} dx, \tag{1.25}$$

$$p_{i+1,j} - p_{i,j} = -q_{i+1/2,j} \left[\frac{h}{2} \left(\frac{1}{\lambda^x_{i,j}} + \frac{1}{\lambda^x_{i+1,j}} \right) \right], \tag{1.26}$$

$$q_{i+1/2,j} = (p_{i,j} - p_{i+1,j}) \left[\frac{h}{2} \left(\frac{1}{\lambda^x_{i,j}} + \frac{1}{\lambda^x_{i+1,j}} \right) \right]^{-1}, \tag{1.27}$$

$$= \frac{2(p_{i,j} - p_{i+1,j})}{h} \left(\frac{1}{\lambda^x_{i,j}} + \frac{1}{\lambda^x_{i+1,j}} \right)^{-1}. \tag{1.28}$$

Since we now have $q_{i+1/2,j}$ expressed as a function of u_i, we can insert the expressions into (1.18). To make the equations clearer, we will use the shorthand

$$\bar{\lambda}^x_{i+1,j} = \left[\frac{h}{2}\left(\frac{1}{\lambda^x_{i,j}} + \frac{1}{\lambda^x_{i+1,j}}\right)\right]^{-1} \tag{1.29}$$

making the flux expression much easier to read:

$$q_{i+1/2,j} = \bar{\lambda}^x_{i+1,j}(p_{i,j} - p_{i+1,j}) \tag{1.30}$$

The reason for selecting this notation isn't just for readability: When it comes to implementing the actual solver, this makes it easy to encapsulate the complexity of the transmissability into a single routine which can be reused without redoing the calculations later.

This choice of notation leads to the conservation for each cell being

$$\begin{aligned}\int_{\Omega_{i,j}} \frac{q}{\rho} = \quad & \bar{\lambda}^x_{i+1,j} \quad (p_{i,j} - p_{i+1,j}) + \bar{\lambda}^x_{i-1,j}(p_{i,j} - p_{i-1,j}) + \\ & \bar{\lambda}^y_{i,j+1} \quad (p_{i,j} - p_{i,j+1}) + \bar{\lambda}^y_{i,j-1}(p_{i,j} - p_{i,j-1}).\end{aligned} \tag{1.31}$$

Since we are interested in generating a linear equation set with $p_{i,j}$ as the unknowns, we can reorder the terms based on the indices:

$$\begin{aligned}\int_\Omega \frac{q}{\rho} = \quad & p_{i,j} \quad \left(\bar{\lambda}^x_{i+1,j} + \bar{\lambda}^x_{i-1,j} + \bar{\lambda}^y_{i,j+1} + \bar{\lambda}^y_{i,j-1}\right) - \\ & \bar{\lambda}^x_{i+1,j} \quad p_{i+1,j} - \bar{\lambda}^x_{i-1,j}p_{i-1,j} - \bar{\lambda}^y_{i,j+1}p_{i,j+1} - \bar{\lambda}^y_{i,j-1}p_{i,j-1}\end{aligned} \tag{1.32}$$

Since we have a total of $n+1$ nodes, we will define $m = n-1$ to simplify the treatment of the inner nodes.

We now see that by ordering the cells in our system in some practical way, we can create a linear system with m^2 unknowns where each cell pressure depends on it's nearest neighbours. By letting the ordering run in horizontal direction from left to right, we can create a vector for the pressure unknowns

$$\mathbf{p}_{i+jm} = p_{i,j} \tag{1.33}$$

and in the same manner define a source term vector from the left hand side of (1.31).

$$\bar{\mathbf{b}}_{i+jm} = \int_\Omega \frac{q}{\rho}. \tag{1.34}$$

The integral over each cell can in practical implementations be solved using a quadrature or for actual physical systems it can be sufficient to have cell wise constant source terms, leaving $\bar{b}$ as a simple reordering of the source terms multiplied with h^2.

Since $\mathbf{p}$ consists of the inner points of the grid, we must add inn the values $\bar{\lambda}^y_{i,j} p_{i,j}$ for defined values of $p_{i,j} \in \partial\Omega$ to $\bar{\mathbf{b}}$ at the correct elements. This will be the first m elements, corresponding to the influence from the upper edge, the last m elements as the influence from below and every m-th element starting from the first and the m-th element to handle the left and right edges. If we denote this sum of Dirichlet boundary conditions and integral source terms as $\mathbf{b}$ we finally have a linear system

$$A\mathbf{p} = \mathbf{b}. \tag{1.35}$$

Since each element is defined by the nearest neighbours, this matrix will have a tridiagonal sparse block structure as shown in in Figure 1.1. In addition, since we assume that transmissability between cells is symmetric, the matrix itself will be symmetric.

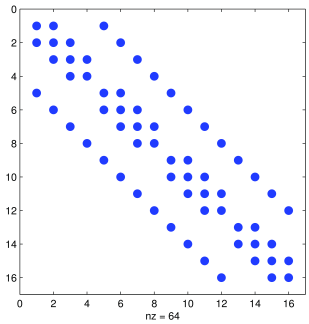


Figure 1.1: Pentadiagonal sparse matrix for the finite volume 2d pressure solver for a 4x4 system

The classical model equation for numerical methods is the Poisson equation. Since our model equation (1.7) is very similar to the Poisson problem

$$\nabla^2 \phi = f \tag{1.36}$$

which could be seen as a special case of (1.7) for constant $\lambda = 1$. This can be a useful sanity check: Since for example the five point stencil created using

finite difference methods gives rise to a symmetric sparse block tridiagonal matrix, we can see that we are on the right path. While this is fairly trivial for the volume method itself, as it has been stated many times before in teaching materials, it can be useful when testing our solver implementation: A sanity check would be to pass inn constant values for λ; If the resulting solution is significantly different from the Poisson solution, something is obviously wrong! It is also worth noting that applying FDM and FVM will both result in the same equation set for this problem: The two perspectives coincide and the difference scheme is conservative.

For equidistant nodes, this scheme is consistent [Aav04],

$$\lim_{h \to 0} \frac{e(u)}{h} \to 0, \tag{1.37}$$

where $e(u)$ is the error from the real solution. It is also stable under the l^2 norm $\|\mathbf{x}\| = \sqrt{\sum_i x_i^2}$, which by the Lax equivalence theorem [LeV07] means that it is convergent.

Chapter 2

The Multiscale Finite Volume method

2.1 The Multiscale approach

While there exist schemes for solving pressure and transport problems to an arbitrary degree of precision, this is often not feasible or practical. A typical physical problem requiring pressure and transport calculations would be an oil reservoir where oil and gas is layered in between different types of rock and sand. The geological complexity can be immense: Changes in rock type can vary on a very small spatial scale, requiring a very high resolution to capture all the details required for fluid transport. Since reservoirs typically can be several kilometers long, it is anything but trivial to compute fluid transport even with methods that always converge to an accurate solution.

To alleviate the need for solving these extremely large systems in full detail, multiscale methods have been developed. The key idea of the multiscale approach in reservoir simulation is that the pressure and transport, while tightly connected, have different requirements. The pressure has weaker dependency on the global changes in permeability than the fluid flow; this can be exploited by making a lower resolution pressure solution before calculating fluid transmission from the resulting pressure. For the MsFV-method, this is done by constructing basis functions for the pressure which try to capture the varia-

tions small scale permeability by solving a series of local problems with different pressure configurations. This is advantageous because solving several small problems may both be quicker and scale better than a single big problem, and because the process can be made highly parallel, optimal for modern technology like multi-core computers, distributed computing and scientific GPU-environments like CUDA where memory is distributed and a high degree of parallelism is required.

2.2 The method summarized

Since we will soon describe the MsFV-method in great detail, it will be useful to have a short statement of the different steps without describing all details:

1. The method first solves many small unit pressure problems distributed over a coarse grid to obtain a pressure basis. Each block covers a subdivision of the fine cells.
2. These basis functions are then used to estimate the flux in another coarse grid, which is shifted compared to the previous grid from unitary pressure so that the old corners become the center points.
3. By applying conservation laws, these fluxes can be used to find the pressure on the coarse grid
4. The basis functions found in step 2 can then be scaled with these pressure values to find a pressure value in all fine cells.
5. Another set of problems can be solved using the flux from the initial pressure solution to get another pressure solution giving conservative fine flow.

2.3 Grids

At the core of MsFV-method, there are several grids which must be fully understood to be able to implement the method:

Definition 1. Ω *is the problem domain, defined as a number of connected fine cells spanning the entire domain.*

Definition 2. *$\bar{\Omega}$ refers to the coarse grid which covers Ω. The coarse blocks each contain a subdivision of the fine underlying grid. On figure 2.1 this is defined by the thick black lines with the corner nodes x_i marked by black squares. For a block $\in \bar{\Omega}$ we will use $\bar{\Omega}_i$*

Definition 3. *$\tilde{\Omega}$ is the dual volumes created by the polyhedron found by taking the geometrical center points of the coarse volumes. On figure 2.1 the dual coarse grid is defined by the blue lines with the corners (primal center nodes) x^k marked in red.*

It is important to note that while the illustrations for simplicity show an equidistant grid with right angles, there are no such restrictions on the method itself.

2.4 Basis functions

The MsFV-method relies on constructing several basis functions which are defined on smaller subproblems of the main problem. The basis function ϕ_i^k is the solution of the reduced homogeneous problem

$$-\nabla \cdot \lambda \nabla \phi_i^k = 0 \text{ on } \tilde{\Omega}_i, \phi_i^k = v_i^k \text{ on } \partial\tilde{\Omega}_i \tag{2.1}$$

For all coarse blocks $\tilde{\Omega}_i$ in the dual coarse grid block $\tilde{\Omega}$. The boundary conditions on the edges/faces of each cell are themselves solutions of the same equation for a lower dimension:

$$-\nabla \cdot \lambda \nabla v_i^k = 0 \text{ on } F \tag{2.2}$$

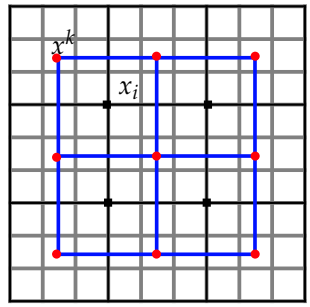
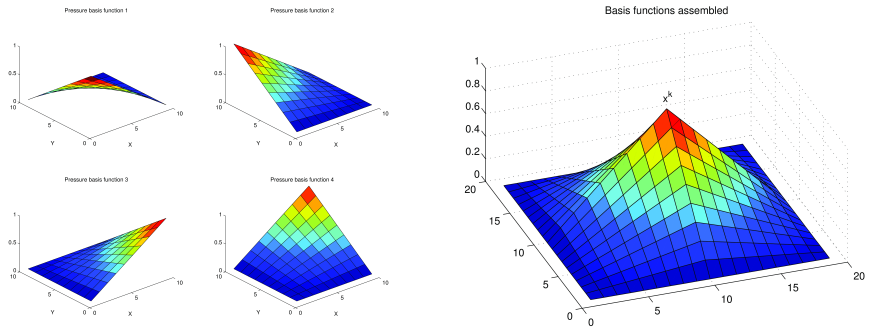


Figure 2.1: Coarse grid and dual coarse grid illustrated.

with the boundary condition $v_i^k(x^l) = \delta_{lk}$ on the edges F. For three dimensions, this requirement becomes a linear interpolation along the edges of each face. The i index refers to the primal grid corner node x_i which is at the center of the current dual cell and the k index refers to the dual grid corner node x^k where pressure is set to one. This implies that we will get two basis functions per dual block in 1D, four in 2D and eight in 3D to cover all corners. The basis functions are defined to be zero outside each dual cell. The lower dimensional problem as the boundary solution was originally proposed by [HW97] in their multiscale finite element (MsFE) method as an alternative to the linear interpolation often used in finite element formulations. As we will see later, having good pressure solutions on the boundaries of the basis functions is important because we will use the pressure basis solutions close to the boundaries to construct mass balance equations to find the global pressure. The MsFV-method is closely related to the MsFE-method described in [HW97], which uses basis functions in a similar manner.



(a) (b)

Figure 2.2: The four basis functions (a) assemble into a single function (b) which vanishes on neighbouring points

While the formalism quickly can get confusing with all the indices, the essence is this: We will find solutions of the homogeneous pressure equation for each dual coarse grid block with unit value at one corner and zero at the others. By doing this for all dual blocks, we eventually arrive at a basis function for each primary grid center point which vanishes at all other center points (compact support in the same manner as FEM elements). Also, since the boundary conditions on the edges overlap from neighbouring dual cell to the next, we have

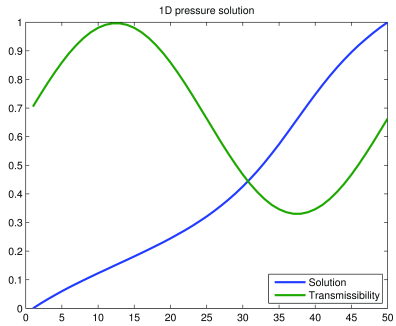


Figure 2.3: 1d solution of the pressure equation to be used as a boundary for the 2d solver.

over the edges continuity for all i for a given k. The basis functions for a single corner point x^k can then be added together to create a continuous function for that specific primal cell center which vanishes at all other coarse nodes. This assembly can be seen visualized in Figure 2.2b if the preceding description is not clear.

For an example of how the pressure basis functions reflect the underlying permeability structure, see how the permeability distribution (along the xy-axis) is reflected in the bumps on the corresponding basis functions above in Figure 2.4. It is also worth noting that for these homogeneous solutions of the pressure equation, the solutions are monotonically decreasing from the high value to the low: If there would be an increase in pressure somewhere in the path from pressure 1 to 0 without any source terms the solution would be unphysical.

2.4.1 Correction functions

While the basis functions can be used to estimate flow from pressure on the coarse nodes, it assumes no-flow along the edges of the coarse blocks induced by any source terms. It also makes an assumption of flow along the edges only depending on the pressure in the neighbouring coarse block centers. This assumption can be violated by for example a high transmissibility value and source terms on the edge of some block. To account for this we can solve the full problem, including source term, on the dual block in almost the same manner

as the basis functions ϕ_i^k:

$$-\nabla \cdot \lambda \nabla \phi_i^k = q \text{ on } \tilde{\Omega}_i, \phi_i^k = \nu^k \text{ on } \partial\tilde{\Omega}_i \tag{2.3}$$

with boundary condition defined in the same way.

$$-\nabla \cdot \lambda \nabla \nu^k = q \text{ on } F. \tag{2.4}$$

Note that instead of setting the boundary condition for ν^k to the Kronecker delta, we simply set it to zero at all corners since the correction function attempts to capture the pressure contribution from the source terms.

2.5 Coarse pressure solver

By observing that the basis functions for each point x_k are continuous when summing over the different neighbouring dual grid cells, we can define

$$\phi^k = \sum_i \phi_i^k. \tag{2.5}$$

We are interested in the pressure solution in the multiscale approximation space $U^{ms} = \text{span}(\phi^k)$. The idea is that the independent pressure basis functions

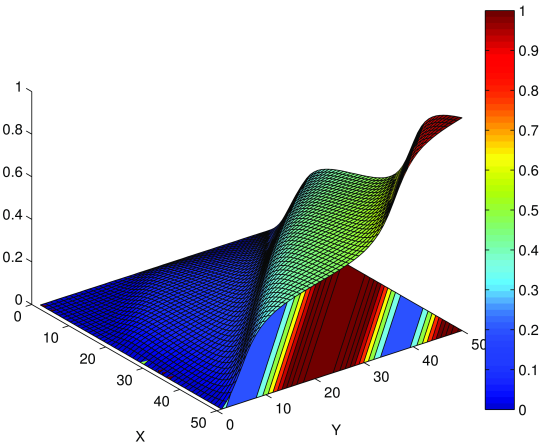


Figure 2.4: The basis function reflects the underlying permittivity changes

capture the influence of the local permeability structure and that for a solution $\{p_k\}$ of the global pressure defined on all primal coarse grid center nodes x_k we can write the solution for all points in the approximation space as

$$p = \sum_k p^k \phi^k = \sum_k \sum_i p^k \phi_i^k. \tag{2.6}$$

Since the basis functions vanish at all but one x_k we see that $\phi^k(x_k) = p_k$, so such a linear combination solution from the coarse solution will have the same values at all the coarse nodes. The basis functions will act as interpolants, giving a gradual transition between various coarse nodes while still reflecting the underlying permeability.

We now have an extension from some coarse solution $\{p_k\}$ to the fine grid for all nodes. The next objective is to obtain equations for finding such a coarse pressure solution. The solution is set up mass balance equations defined by (1.18) for every primal cell as induced by the pressures $\{p_k\}$ when applied to the basis functions. The index for some cell in the primal coarse grid $\bar{\Omega}$ will be l, following convention from [LAKK].

We can formulate the flux out of primary coarse grid block B_l induced the pressure differential over the block edge by basis function ϕ^k as

$$f_{k,l} = \int_{\partial B_l} n \cdot \lambda \nabla \phi^k ds. \tag{2.7}$$

This again leads to the mass balance equations for each cell B_l

$$\sum_k p^k f_{k,l} = \int_{B_l} q. \tag{2.8}$$

There is one unknown p_k for each block on the primal grid so that we have a sufficient number of equations to determine the pressure for all the coarse nodes.

2.6 Interpolating the coarse pressure field

Once we have solved the small linear equation set for the pressure, we can use the basis functions to find pressure values in all fine cells simply by adding together the basis functions multiplied with the corresponding coarse pressure

values. Since the basis functions ϕ^k capture the local variations in transmissibility in their local domain, the idea is that this linear combination will give a good approximation of the much more computationally intensive solution on the fine cells.

2.7 Fine flux reconstruction

The reconstructed pressure will be continuous and the pressure solver is fully stand alone. However, while the reconstructed pressure is continuous, it will not be smooth; A naive attempt to calculate fluid velocity using some approximation of

$$\mathbf{u} = -\lambda \nabla p \tag{2.9}$$

may result in non-conservative flow across the edges of dual grid blocks at the fine level since the pressure only has piecewise continuous derivative, with a discontinuity at the edges. This is a problem because the primary motivation for using a finite volume method its conservative properties.

We can construct a new set of basis functions, commonly referred to as the fine scale basis functions, to resolve this fine flow. This consists of simply solving the full problem (1.7) in each primal coarse block $\bar{\Omega}_k$ with the flux from the original pressure solution as the boundary condition. Source terms can be handled in different ways, one of which being to simply distributing the source terms inside the block equally. This leads to another set of pressure values which coincide along the edges of primal cells and can be used with some discretization of (2.9) to find the flow. While this new pressure makes the flux continuous, it can theoretically make the pressure discontinuous, so this step should only be done if we are interested in the fluid velocity.

2.8 Variants

Since the MsFV-method was introduced in 2003, there have been a multitude of papers published on the method, its advantages and disadvantages. The method has also been extended to overcome weaknesses and variants proposed to handle different types of problems. This section summarizes some of the more important developments.

2.8.1 Iterative and adaptive formulation

The biggest weaknesses of the original formulation [JLT03] are the error arising from the decoupling of the local problems and the basis functions' inability to handle complex flow from small wells and other features not captured by the basis functions. The assumption is that the reduced one dimensional problem along the edges will be sufficient to estimate the pressure along the edges. However, this decoupling fails for problems solved in highly anisotropic heterogeneous mediums [LJ07]. Intuitively this isn't hard to understand: If the pressure problem could be approximated well by a series of lower dimensional problems everywhere, the influence from changes in pressure would only be constrained to a small local neighbourhood, something which isn't the case. There are, however, ways of overcoming this problem: In [HJ11] a solution was proposed where the correction functions remove errors in the solution. By using an iterative approach, parts of the pressure field not captured by the basis functions can be estimated and added to the solution.

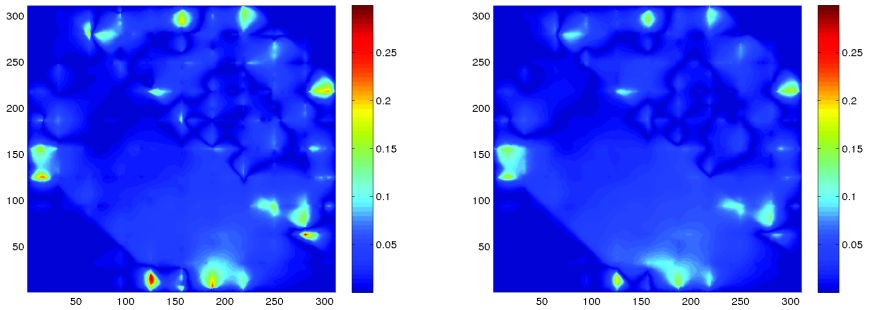
The central idea is to calculate the correction functions normally during the first iteration and when the pressure solution has been assembled apply a smoother of the form

$$u_{j+1} = u_j + B(f - Au_j). \tag{2.10}$$

This smoother will typically be some cheap iterative linear solver (Jacobi, Gauss-Seidel) which could theoretically solve the full problem with sufficient iterations. While the solver *could* be used to solve the entire linear system for the full problem with a large amount of iterations, it is instead applied a few times to the MsFV solution to smooth out the solution and correct the pressure along the edges of the dual blocks as shown in Figure 2.5. Since the error will be greatest on the edges of each $\tilde{\Omega}_i$, a few iterations of a smoother can improve the pressure in those areas based on the neighbours fairly quickly.

When the smoother has been applied, this solution is then used as a boundary condition for a new set of correction functions. The coarse pressure can then be solved with the new correction flux, without the need to regenerate anything but the **b** vector since the correction functions only influence the right hand side of 2.8. The smoother can then be reapplied and the process repeated to get gradually better solutions. This approach was first described in [HBHJ08].

This approach makes the MsFV-method into something closely resembling the successful multigrid methods used for similar problems, described in [Saa03], which also use both smoothing, restriction (coarse system) and prolongation (here described as the interpolation) steps.



(a) Before iterating

(b) After 1 iteration

(c) After 5 iterations

(d) After 50 iterations

Figure 2.5: The iterative MsFV-method reduces error on coarse cell edges. Figure from the pre-master project.

Since these iterations are dominated by the computations required to recompute the basis functions in each step, it can be useful to calculate which basis functions need to be recomputed. In [HJ11] some methods of deciding which functions to update are described.

One thing to consider is to calculate the residual of the solution,

$$\mathbf{R} = \mathbf{A}\mathbf{x_i} - \mathbf{b}, \tag{2.11}$$

where $\mathbf{x_i}$ is the solution vector after step i. This can then be used to identify

fine cells with large errors using

$$\frac{|r_i|}{\|\mathbf{R}\|} > \epsilon_a. \tag{2.12}$$

This can also be applied to the right hand side of the coarse pressure system 2.8 ($\mathbf{q}$)

$$\frac{1}{1+\epsilon_b} < \frac{q^n}{q^*} < 1+\epsilon_b, \tag{2.13}$$

and to the mobility λ

$$\frac{1}{1+\epsilon_b} < \frac{\lambda^n}{\lambda^*} < 1+\epsilon_b. \tag{2.14}$$

In both the preceding equations n denote the current state and * the previously updated state. Note that the consideration of the mobility is only required in multiphase systems where the relative permeability of a cell can change.

Once we have identified these critical fine cells, we can then update as needed the corresponding correction functions (and basis functions for the multiphase case) during the next iteration. If an block contains a critical fine cell, the corresponding functions are updated.

2.8.2 Extension to multiphase problems

The MsFV-method is fairly straight forward to extend to multiphase problems. The main difference from the single phase model is that the basis functions must be regenerated every time step, because the local relative permeability depends on the different phases. [HJ09, JLT05]. While this has an increased computational cost, multiphase problems are in general more computationally intensive.

Chapter 3

Operator formulation of the Multiscale Finite Volume method

3.1 Introduction

An interesting development in research on the MsFV-method is the formulation of the method as a set of operators on an linear equation set described described in [ZT08], [NB08] and [LTL10]. This version formulates the method as a set of operators which act upon the full linear system created using some discretization of the pressure problem, and not as a method that solve the pressure problem directly. This formulation is especially interesting since it makes the implementation and deployment of the multiscale solver into an already existing framework easier: The sometimes very complex code used for generation of equation sets for commercial reservoir simulation software does not have to be modified to enable a multiscale solver - rather, a new solver is implemented which acts on the linear equation sets generated, without much knowledge on how the systems themselves are generated.

The operator formulation itself is just another perspective on the same method, but it has some useful observations about the linear algebra involved, for exam-

ple how the resulting equation sets can be efficiently solved using the Generalized Minimal Residual method (GMRES). In addition to this, it makes extension to complex grids much easier if a dual grid can be created. For this thesis, the goal is to adapt the operator formulation to the MATLAB Reservoir Toolbox (MRST).

The notation and treatment here mirrors [ZT08] and [LTL10] closely, while aiming to make the material more available by being more explicit than the articles have room for.

3.2 Notation and preliminaries

The operator formulation starts with some linear equation set

$$A\mathbf{u} = \mathbf{r}, \tag{3.1}$$

similar to the one described in Section 1.2. This equation set typically has the pressure as the unknown $\mathbf{u}$ and a right hand side $\mathbf{r}$ describing the influence from source terms and boundary conditions on the mass balance. The matrix A represents the influence from the different cells on each other.

The key insight in the operator formulation is that if we categorize unknowns according to whether they are on coarse center nodes, internal and inner nodes (see Figure 3.1) of the dual grid, the rest of the method implementation does not need any geometrical understanding of the geometry. In the original formulation, the algorithm must revisit the geometry numerous times in assembling and applying the basis functions, which can lead to many special cases and difficult implementation. In the operator formulation, this is handled by letting the topological information in the system matrix A handle most of the geometry.

Inner, edge and center nodes will be denoted by indices i, e and n respectively. Center nodes refer to the geometric center nodes of the primal coarse blocks, edge nodes are the nodes situated on the edge of each dual block and inner nodes are the rest of the nodes. All nodes can be categorized according to this partitioning scheme,

$$\mathcal{I}_f = \mathcal{I}_n \cup \mathcal{I}_e \cup \mathcal{I}_i. \tag{3.2}$$

3.3 Permuting the system and breaking symmetry

An important part of the operator formulation is the use of permutation matrices.

Definition 4. *A permutation matrix consists of row vectors e_i where element j is δ_{ij} and each e_i is unique:*

$$P = \begin{bmatrix} \mathbf{e}_{i_1} \\ \mathbf{e}_{i_2} \\ \vdots \\ \mathbf{e}_{i_n}. \end{bmatrix} \tag{3.3}$$

Since each vector e_i is unitary and only non-zero in position i, for some arbitrary vector $\mathbf{v} \in \mathbf{R}^n$ $P\mathbf{v}$ has the same values as $\mathbf{v}$: The elements are simply moved to other positions (the elements need not move, as by our definition the identity matrix is an example of a permutation matrix). This can be seen intuitively by appealing to the algorithm of matrix-vector multiplication:

$$(A\mathbf{v})_i = \sum_{k=1}^{n} A_{ik}\mathbf{v}_k, \tag{3.4}$$

which with $A = P$ would become

$$\sum_{k=1}^{n} P_{ik}\mathbf{v}_k = \sum_{k=1}^{n} \delta_{kj}\mathbf{v}_k = \mathbf{v}_j, \tag{3.5}$$

such that each row P_i picks one element from $\mathbf{v}$ and places it in place i. A well known result is that permutation matrices can be inverted by transposing: $P^{-1} = P^T$.

When solving systems of linear equations in PDE applications, the ordering of the points is fairly arbitrary. If nodes are ordered left to right or top to bottom according to the topology should not affect the solution of the linear system[1]. This can be shown by applying a permutation matrix to the left hand side of (3.1).

$$PA\mathbf{u} = PAP^{-1}P\mathbf{u} = PAP^TP\mathbf{u} = P\mathbf{r} \tag{3.6}$$

If we then denote a specific permutation matrix as $\tilde{P}$ and $\tilde{P}A\tilde{P}^T = \tilde{A}, \tilde{P}\mathbf{r} = \tilde{\mathbf{r}}$ and $\tilde{P}\mathbf{u} = \tilde{\mathbf{u}}$ we arrive at a permuted system

$$\tilde{A}\tilde{\mathbf{u}} = \tilde{\mathbf{r}} \tag{3.7}$$

[1]It can however sometimes alter the structure of the system matrix in a way that is beneficial for linear solvers

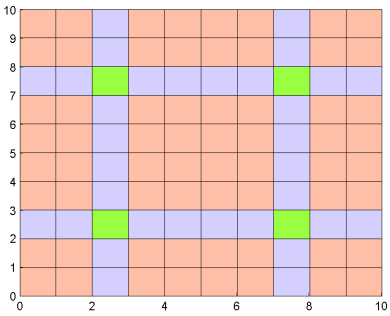
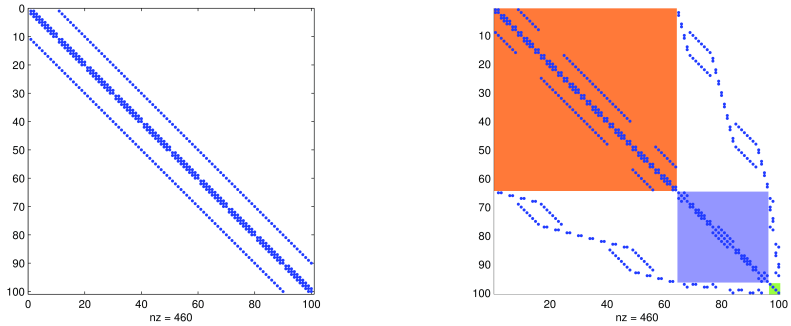


Figure 3.1: The example grid showing inner (orange), edge (blue) and center (green) cells



(a) (b)

Figure 3.2: The original system using left-right ordering (a) is permuted according to cells' status as inner (orange), edge (blue) and center (green) (b) based on the grid in 3.1

In our case, $\tilde{P}$ will be defined so that the system is reordered according to (3.2). This will result in a system where the inner nodes are ordered first, then the edge nodes and last the central nodes. Since the same equations define the same nodes in the permuted system, this has some interesting implications for

the permuted system matrix,

$$\tilde{A} = \begin{bmatrix} \tilde{A}_{ii} & \tilde{A}_{ie} & 0 \\ \tilde{A}_{ei} & \tilde{A}_{ee} & \tilde{A}_{en} \\ 0 & \tilde{A}_{ne} & \tilde{A}_{nn} \end{bmatrix}. \tag{3.8}$$

The first subscript of each block here signifies which category influences the mass balance equation and the seconds subscript indicates what category is being influenced. For example, the block $\tilde{A}_{ie}$ contains the influence the inner node pressures have on the edge nodes' pressure. Note that the zero blocks arise from the assumption of a five point stencil where no central nodes influence inner nodes and vice versa.

Currently (3.8) represents a system where nodes influence each other in a fully symmetric way. The first step towards the operator formulation is to break this symmetry by removing the influence of the inner nodes on the edge nodes and similarly make central nodes unaffected by edge nodes ($A_{ei} = 0, A_{ne} = 0$). This mirrors the approach in Section 2.6, where the algorithm extrapolates from the coarse pressure solution using basis functions constructed from the edge pressure. Of course, this symmetry breaking is not cost free; It places higher importance on some nodes relatively to others and as we will see, for highly anisotropic mediums and difficult geometries this can be problematic for the quality of the solution [2]. The reordered system is then

$$M = \begin{bmatrix} \tilde{A}_{ii} & \tilde{A}_{ie} & 0 \\ 0 & M_{ee} & \tilde{A}_{en} \\ 0 & 0 & M_{nn} \end{bmatrix}. \tag{3.9}$$

Note that the diagonal blocks of self influence for edge nodes and central nodes has been replaced by a new set of matrices M_{ii} and M_{ee}.

M_ee is defined by taking the influence on the edge nodes from the inner nodes, $\tilde{A}_{ie}$ and removing it:

$$M_{ee} = \tilde{A}_{ee} + \text{diag}\left[\sum_i \tilde{A}_{ie}^T\right], \tag{3.10}$$

which ensures that there is no mass balance loss to equations no longer in the system. Following convention from [ZT08], diag is an operator transforming a vector to a diagonal matrix. M_{nn} is a special case, as it forms the linear system corresponding to the coarse pressure system, as in Section 2.5.

[2] For an example of a highly anisotropic case, see the Upper Ness layers in Section 7.6.1

(3.9) will be denoted as the MsFV matrix and can be understood to define a new linear system

$$M\tilde{\mathbf{u}} = \mathbf{q} = \begin{bmatrix} \mathbf{q_i} \\ \tilde{\mathbf{r_e}} \\ \mathbf{q_n} \end{bmatrix}. \tag{3.11}$$

This redefinition as a new system is not strictly needed to implement a pressure solver, but it will be useful when implementing the method as a iterative method to increase the accuracy of the method.

3.4 Solving the decoupled system

Because this system is upper block triangular we can, without any prior knowledge of the algorithm, use block elimination to find an explicit inverse. However, to make the connection between the previous formulation of the algorithm and the operator formulation, we will split the inverse in two parts: One part for the block which will influence $[0\ 0\ \mathbf{q_n}]^T$ and one part for the influence on $[\mathbf{q_i}\ \tilde{\mathbf{r_e}}\ 0]^T$. This is because the solution for the coarse nodes, $\mathbf{q_n}$ is an important step of the algorithm.

If we name these components of the inverse B' and C respectively we get

$$MM^{-1} = M(C + B') = I \tag{3.12}$$

$$\begin{bmatrix} \tilde{A}_{ii} & \tilde{A}_{ie} & 0 \\ 0 & M_{ee} & \tilde{A}_{en} \\ 0 & 0 & M_{nn} \end{bmatrix} \left(\begin{bmatrix} a & d & 0 \\ b & e & 0 \\ c & f & 0 \end{bmatrix} + \begin{bmatrix} 0 & 0 & g \\ 0 & 0 & h \\ 0 & 0 & i \end{bmatrix} \right) = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & I \end{bmatrix} \tag{3.13}$$

If we first solve for B' we get the equations

$$M_{ee}h + \tilde{A}_{en}i = 0 \tag{3.14}$$

$$\tilde{A}_{ii}g + \tilde{A}_{ie}h = 0 \tag{3.15}$$

$$M_{nn}i = I \tag{3.16}$$

Solving this system is a simple process of backwards substitution and gives

$$B' = \begin{bmatrix} 0 & 0 & \tilde{A}_{ii}^{-1}\tilde{A}_{ie}M_{ee}^{-1}\tilde{A}_{en} \\ 0 & 0 & -M_{ee}^{-1}\tilde{A}_{en} \\ 0 & 0 & I_{nn} \end{bmatrix} M_{nn}^{-1} \tag{3.17}$$

Solving for C is also mostly trivial:

$$\tilde{A}_{ii}a + \tilde{A}_{ie}b = I \tag{3.18}$$
$$\tilde{A}_{ii}d + \tilde{A}_{ie}e = 0 \tag{3.19}$$
$$M_{ee}b = 0 \tag{3.20}$$
$$M_{ee}e = I, \tag{3.21}$$

which gives

$$C = \begin{bmatrix} \tilde{A}_{ii}^{-1} & -\tilde{A}_{ii}^{-1}\tilde{A}_{ie}M_{ee}^{-1} & 0 \\ 0 & M_{ee}^{-1} & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3.22}$$

If we use the assumption that B only influences $\mathbf{q_n}$ we can write the solution of (3.11) as

$$\tilde{\mathbf{u}} = \underbrace{BM_{nn}^{-1}\mathbf{q}_n}_{\text{Interpolate coarse solution}} + \underbrace{C\mathbf{q}}_{\text{Correction functions}} \tag{3.23}$$

When armed with knowledge of the MsFV-method in its original formulation, it becomes apparent that this corresponds to solving a coarse system $M_{nn}^{-1}\mathbf{q}_n$ for pressure values in the coarse nodes and then interpolating the solution using the interpolation matrix B, which is reflected in the structure of B itself: It maps coarse pressure to the coarse nodes using an identity matrix, and then maps central nodes' influence to the edge pressure, which is then reused to extrapolate to the inner nodes. C represents the correction functions and handles handles effects not captured by the basis functions, for instance complex behavior from wells. The difference between the correction functions and the interpolated solutions can be seen in Figures 3.3a, 3.3b and 3.3c.

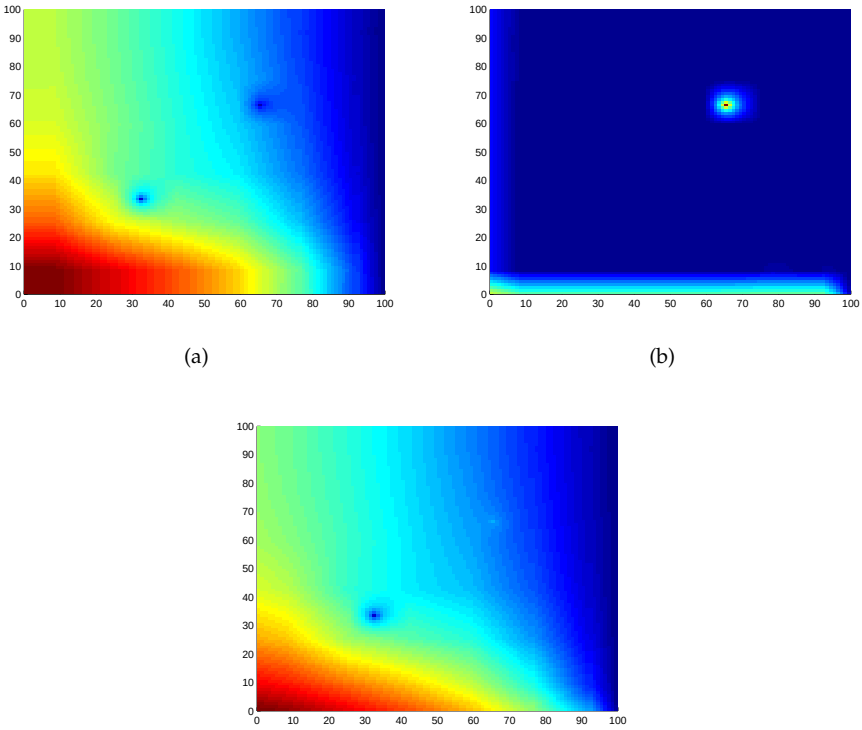
3.5 Coarse pressure system

To solve the coarse system

$$M_{nn}\tilde{\mathbf{u}}_n = \mathbf{q}_n, \tag{3.24}$$

we need a way to assemble the flux contribution across the edges of primal coarse blocks as done in Section 2.5 into a right hand side $\mathbf{q}_n$ as well as find M_{nn}.

To do this, we define the control volume summation operator χ. χ has one row for each block $\bar{\Omega}_i$ and does the summation of values corresponding to fine cells



(a) (b)

(c)

Figure 3.3: The interpolated (a) and corrected (b) solutions of a problem with mixed boundary conditions and two pressure wells combine to the final solution (c).

inside this block. To achieve this, it is defined as

$$\chi_{ik} = \begin{cases} 1 & \text{when } x_k \text{ is inside } \bar{\Omega}_i \\ 0 & \text{otherwise} \end{cases} \tag{3.25}$$

When applied to a vector each row can be seen as looping over all cells and picking the values from those inside that rows coarse block. As all operators, it may or may not be beneficial to implement it as an actual matrix.

Taking (3.23) and inserting for $\tilde{\mathbf{u}}$ in (3.11) while applying the control volume

summation operator we get

$$\chi\tilde{A}\tilde{\mathbf{u}}_n = \chi\tilde{A}B\tilde{\mathbf{u}}_n + \chi\tilde{A}C\mathbf{q} = \chi\tilde{\mathbf{r}} \tag{3.26}$$

Since this is the same mass balance we formulated in (3.24) we can select

$$M_{nn} = \chi\tilde{A}B, \tag{3.27}$$

and finally by rearranging the terms so M_{nn} is on the left hand side;

$$M_{nn}\tilde{\mathbf{u}}_n = \chi\tilde{\mathbf{r}} - \chi\tilde{A}C\mathbf{q} = \mathbf{q}_n. \tag{3.28}$$

When we have a solution $\tilde{u}$ it is important to permute back to the original ordering to make sense of the results for plotting and analysis: Since the inverse of a permutation matrix is its transpose, we simply write $\mathbf{u} = \tilde{P}^T\tilde{\mathbf{u}}$

This is sufficient for the pressure solver, but an additional step is needed for finding a suitable field for flux construction. Note also that in Section 3.7 additional steps for an iterative smoother is presented.

3.6 Constructing a conservative fine flux field

To reconstruct a conservative fine flow field, we must produce another permutation matrix $\bar{P}$. $\bar{P}$ orders the unknowns based on which primal coarse block $\bar{\Omega}_i$ it is in. All unknowns corresponding to block i will come in order and again we define a permuted system

$$\bar{A} = \bar{P}A\bar{P}^T \tag{3.29}$$

$$\bar{\mathbf{u}} = \bar{P}\mathbf{u} \tag{3.30}$$

$$\bar{\mathbf{r}} = \bar{P}\mathbf{r}. \tag{3.31}$$

An example of $\bar{A}$ can be seen in Figure 3.4 with green marking the coarse blocks. The idea is to use Neumann boundary conditions on the coarse primal elements based on the pressure solution in the previous section. Since the coarse problem (3.24) is constructed to ensure mass balance between each $\bar{\Omega}_i$, the system is conservative over the coarse blocks. From this solution we can then construct a fine conservative flow by taking the current flux over each primal boundary as the boundary condition for each local problem. Since the discretization is assumed to be a FVM discretization leading to conservative flow, the resulting solution will be conservative at all cells. However, as the

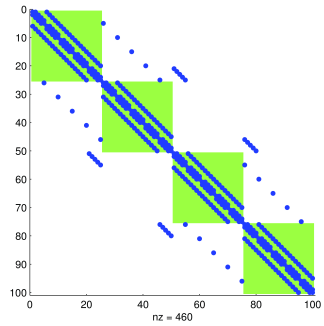


Figure 3.4: The system permuted by $\bar{P}$ so that the grid is lexicographic in the primal blocks

boundary is strictly a restriction on the magnitude of the flux and not the pressure value itself, the reconstructed solution is not guaranteed to be a continuous in terms of pressure.

For this operation, we will need the block diagonal part of $\bar{A}$ which will be referred to as $\bar{A}^D$. Each diagonal block represents the mass balance contribution of cells inside some primal block $\bar{\Omega}_i$ on other cells within the same block. Analogously to the treatment of $\tilde{\Omega}$, off-diagonal blocks refer to the contribution from coarse blocks $j \neq i$ and will be removed from the system. To remove this influence, we must alter the diagonal of $\bar{A}^D$ so that the nodes from different coarse blocks have no more influence:

$$D = \bar{A}^D - \text{diag}\left\{\sum_j \bar{A}_{jk} - \bar{A}^D_{jk}\right\}. \tag{3.32}$$

In other words, we subtract the sum of the off-diagonal elements from the diagonal. Note that while [ZT08] and [LTL10] uses a plus sign in (3.32), this will lead to the wrong results when implemented.

This leads to a new problem,

$$D\bar{u} = \bar{r} - (\bar{A} - D)\bar{P}\mathbf{u}. \tag{3.33}$$

where the right hand side is local problems for each primal block $\bar{\Omega}_i$ and the right hand side represents Neumann boundaries leading to continuous flux using the pressure solution.

Note that this step is optional and is not required when only interested in the pressure solution.

3.7 Multiscale iterations

Since the solution of the multiscale solution in general does not coincide with the full solution, there is some interest in being able to improve the solution to an arbitrary degree of precision without incurring a heavy computational cost. The operator formulation is no exception, and in [LTL10] several approaches for converging to the correct solution were considered. We will consider both MsFV-iterations and one of the more promising smoothers.

3.7.1 MsFV iterations using GMRES

To construct an iterative method from the MsFV-formulation, a new solution should have source terms corresponding compensating for the localization error made when formulating the method. The first step is to formalize the right hand side of the multiscale system. The first terms, corresponding to the inner and edge nodes are the same as for the original system $\tilde{\mathbf{r}}$. The last terms in the vector, however, will be the entries of $\mathbf{q}_n$ which correspond to a flux integral over the edges of each $\bar{\Omega}_i$. To create the resulting vector, we will use $R = [00I_{nn}]$, an operator with the following properties:

$$R^T \begin{bmatrix} \mathbf{V}_{ii} \\ \mathbf{V}_{ee} \\ \mathbf{V}_{ii} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \mathbf{V}_{ii} \end{bmatrix} \tag{3.34}$$

$$R^T R = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & I_{nn} \end{bmatrix} \tag{3.35}$$

It is then straightforward to define an operator Q so that

$$\mathbf{q} = Q\tilde{\mathbf{r}}, \tag{3.36}$$

by using the properties of R along with the definition of $\mathbf{q}_n$ from Equation 3.28 to get

$$Q = \underbrace{I - R^T R}_{\text{Identity except the last n elements}} + \underbrace{R^T(\chi - \chi\tilde{A}C).}_{\text{Gathering the last n from } \mathbf{q}_n} \tag{3.37}$$

Where n is the number of inner nodes. This operator transforms any right hand side terms for the original system into an appropriate right hand side vector for multiscale operator M.

If we have some solution $\tilde{u}^\nu$ we want to improve, we can then add the residual error as a new source term to get an improved solution $\tilde{i}^\mu$,

$$\tilde{u}^\mu = \tilde{u}^\nu + M^{-1}Q(\tilde{r} - \tilde{A}\tilde{u}^\nu). \tag{3.38}$$

In (3.38) the residual is the difference between the original right hand side and the original system matrix multiplied with multiscale pressure solution. This is a measure of the error inherit in the multiscale method when compared to the original system, which is added to the system as a new multiscale solution. This is potentially a fixed point iteration scheme when $\tilde{i}^\mu = \tilde{i}^{\nu+1}$:

$$\tilde{u}^{\nu+1} = \tilde{u}^\nu + \omega M^{-1}Q(\tilde{r} - \tilde{A}\tilde{u}^\nu). \tag{3.39}$$

We have here added an relaxation parameter ω which can be used to stabilize the iterations by letting the increments be sufficiently small. To avoid delving into iterative theory, we will simply note that this is a Richardson iteration with preconditioner $M^{-1}Q$. As noted in [LTL10], a better alternative is to solve the problem using the Generalized Minimal Residual Method (GMRES), which is both faster and converges monotonically to the exact solution.

3.7.2 Iterative smoothing

Instead of using the multiscale formulation for iterations, we can instead successively apply a smoother. As mentioned in Section 2.8.1, the errors are oscillatory near the edges of coarse volumes and a smoother will then be able to quickly remove large amounts of the errors. The resulting pressure with the highly oscillatory errors removed can then be reused for another MsFV iteration. [LTL10] suggests that the Dirichlet Multiplicative Schwarz (DMS) algorithm gives good performance, but in principle any kind of cheap smoother can be used.

The idea is to use the system permuted by the ordering in the primal blocks as in Section 3.6 to construct $\bar{A}^D$ and $\bar{A}^U$, where $\bar{A}^U$ is the upper block diagonal part of $\bar{A}$. We then get the DMS iterations,

$$\bar{u}^{\nu+1} = \bar{u}^\nu + \omega(\bar{A}^U + \bar{A}^D)^{-1}\bar{u}^\nu. \tag{3.40}$$

As well as the Dirichlet Additive Schwarz (DAS) iterations,

$$\bar{u}^{\nu+1} = \bar{u}^{\nu} + \omega(\bar{A}^{U})^{-1}\bar{u}^{\nu}. \tag{3.41}$$

These iterations are obviously cheap; The system $\bar{A}^{U} + \bar{A}^{D}$ is upper block Hessenberg and fairly inexpensive to solve.

Chapter 4

The MATLAB Reservoir Simulation Toolbox

4.1 Introduction

Actual realization of numerical models for reservoir simulation is a very complex task. The software should ideally simultaneously implement modern methods, quickly give solutions with a high degree of precision and be easy to use. It is hard to achieve all these things at the same time, and choices will be made. In this section some of the various standards will be presented, as well as the reasoning behind the choices made in the Matlab Reservoir Toolbox which will be used for a prototype implementation.

4.2 Types of grids

4.2.1 Cartesian grids

Uniform Cartesian grids are the most common type of grids in academic research due to their ease of implementation for most methods. They consist of equidistant nodes in each direction, forming squares in 2D and cubes in 3D.

The grid spacing can also vary, for example if adaptivity is needed.

A simple Cartesian grid is shown in Figure 4.1.

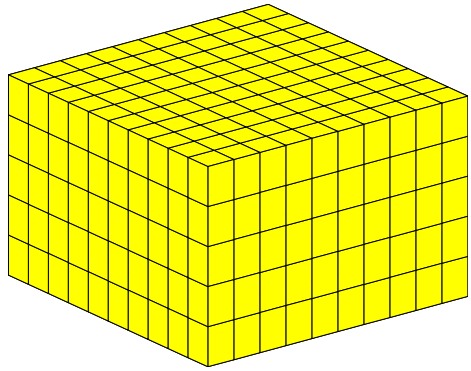


Figure 4.1: An Cartesian grid with uniform grid spacing

4.2.2 Corner point grids

Corner point grids are the de facto standard for reservoir simulation, and can be seen as a midpoint between cartesian grids and fully unstructured grids. In corner point grids, the grid consists of "pillars" on which the corners of all cells in the grid are defined. The pillars are ordered by their spatial position. This gives much more flexibility than Cartesian grids when it comes to representing for example faults.

An example, including a fault is shown in Figure 4.2.

4.2.3 Unstructured grids

Fully unstructured grids explicitly define cells by their points and neighbouring structure. This means that any grid can be approximated closely, with the only restriction being linear polynomials used for the edges. An example can be seen in Figure 4.3.

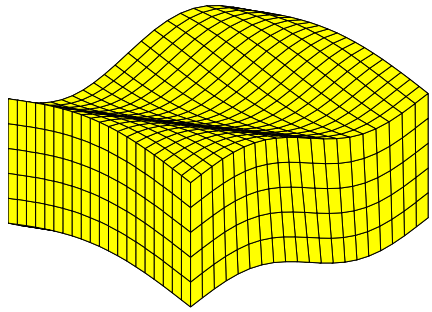


Figure 4.2: A corner point model with fault

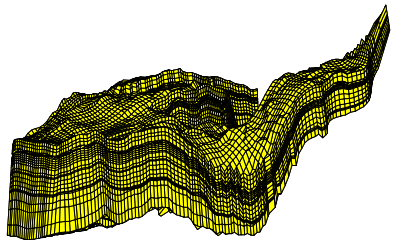


Figure 4.3: A corner point model with fault

4.3 Grid and geometry representation in MRST

For the Matlab Reservoir Toolbox (MRST) the developers chose to represent all grids in the same normalized format. No matter how complex the structure may be, the data format is fundamentally the same. This has the advantage of making it easy to implement a method for different grids: If the method works on the unstructured grids in 2D and 3D, it will work for all the different grids in the toolbox.

The downside of this choice is of course that there is a large overhead to representing any grid in a fully general data format. For instance, if we are to create a 2D grid with NxN equidistant nodes we will require storage for all the node positions, storage for all the cell faces in the system and the cells themselves. Each cell will need storage for indices of its four faces, each face must store

the indices of two nodal points and each node must store a tuple (x, y) for the coordinates. In addition, a lot of additional data will be generated, like the neighbour structure of the faces. If we were to just store the essential information required to represent this grid, just the distance between two nodes, the number of nodes in each direction and the coordinates of one corner.

The overhead is smaller for more complex grids and this approach has the advantage of making new methods easier to prototype as only one grid datastructure should be considered. A typical grid in MRST will be a struct, which will have the following fields:

- `cells` containing cell data
- `faces` containing face data
- `nodes` containing nodes data
- `cartDims` shows, for Cartesian grids, the cell count along different axes. Is a vector of length 2 and 3 in 2D and 3D respectively.
- `type` gives hints for the type of grid and what operations has been run on it, for instance of face normals have been calculated.
- `griddim` is the dimension of the grid, a scalar 2 or 3.

All geometrical data builds upon the `nodes` structure, which is defined as

- `num` - a scalar describing the number of total nodes in the model
- `coords` - a `num` $\times$ `griddim` matrix where row i contains the coordinates of point i.

The `nodes` contain all the positions in space for the grid. The `faces` struct builds upon this, and relies on `nodes` for spatial positions.

- `num` gives the number of total faces in the model
- `nodePos` is an *indirection map* into the `faces.nodes` array. This means that to lookup the coordinates of face `i`, we must access `faces.faces(faces.nodePos(i):faces.nodePos(i+1)-1)` to find the indices.

- `neighbors` is an array `num`$\times$2 long, where row `i` gives the indices of the neighbouring cells of face `i`. 0 is given where the face has no neighbor, for example on the edge of the domain.
- `tag` no current official usage.
- `nodes` - see `nodePos` for usage.
- `normals` contains the normal vectors for face `i` in row `i`. The orientation of the normals when applied to a cell depend on the position of the cell in the `neighbor` array.
- `centroids` contains the centroid for face `i` in row `i`.
- `areas` contains the area of face `i` in element `i`.

And again we can use the definition of the faces to create 3D cells, which are defined in the `cells` struct:

- `num` gives the number of total cells in the model
- `facePos` is an *indirection map* into the `cells.faces` array. The usage is analogous to the `nodePos` in `faces`.
- `indexMap` maps active cells to global cell indices.
- `faces` - this is where indices from `facePos` can be used to lookup actual face indices.
- `volumes` contains the volume of cell `i` in row `i`.
- `centroids` contains the centroid for cell `i` in row `i`.

With variable number of faces per cell and nodes per face, this can be used to construct any general grid under the assumption that any curved surfaces can be accurately approximated using piecewise linear polynomials. MRST has utilities for importing grids in common data formats, like `.grdecl`, the Eclipse data format.

Chapter 5

Implementation of the MsFV-method

5.1 Implementation in the earlier project

Before writing this master's thesis, a project was performed to gain a better understanding of the method along with a literature study. The implementation was done from the ground up in MATLAB, with two-point flux approximation of permeability and only equidistant 2D grids being used. The implementation used the original formulation, without any linear algebra required.

5.2 Primal grid

The first component needed to create a MsFV-implementation is a coarse grid. There are several approaches for creating a coarse grid, but forunately for us, MRST already has a coarse grid module which supports different partitioning schemes for creating a coarse grid. We will use the existing routines for this purpose.

The coarse grid module in MRST defines coarse grids in the same manner as a fine grid. While not every aspect of a fine grid is duplicated - for instance,

nodes for the coarse grid is implicitly defined via the fine grid - a lot of useful information is contained in the module, for example routines for calculating centroids, volumes and face areas.

5.2.1 Partition uniformly

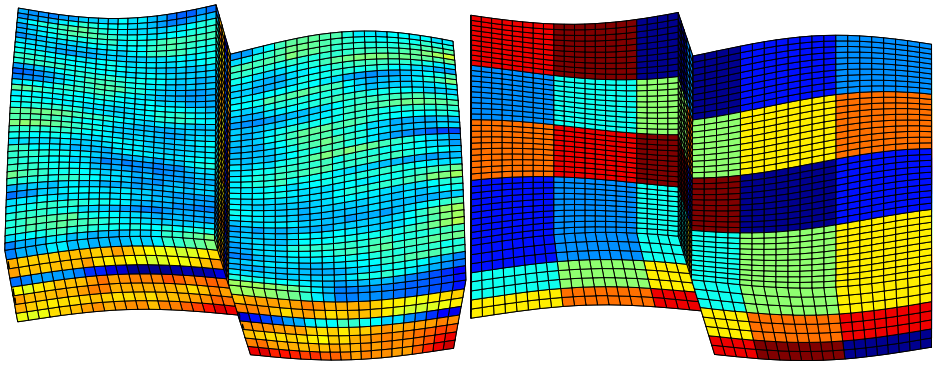
The simplest partitioning scheme is the uniform partitioning scheme, called by `partitionUI` which generates a coarse grid with a given number of coarse volumes along each axis of the logical coordinates i, j, k. The resulting partitions are approximately equal in size, following a load balanced scheme for distributing the fine cells. An example is shown in Figure 5.1b.

5.2.2 Partition by layers

Multiscale solutions usually aim for some degree of homogeneity inside each coarse block. To achieve this, a uniform partitioning scheme is not always suitable. For example, when faced with a layered permeability distribution such as the one found in sedimentary rocks, it can be useful to let the natural layers disconnect each set of coarse blocks. An example of this can be seen in Figure 5.1c, where the function `partitionLayers` is used to partition the grid with the permeability distribution shown in Figure 5.1a into different layers.

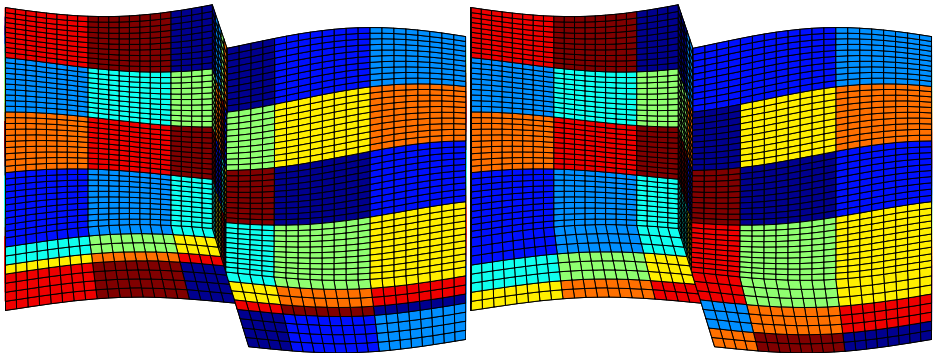
5.2.3 Process partition

Partitioning the grid using logical indices is useful because it is fast and geometry independent, but since blocks which are logically next to each other can be physically separated near for example faults, MRST provides a routine to handle this situation. For example, the coarse grid in Figure 5.1c has coarse blocks which are divided in two by the fault, but are still categorized as the same block. This can be seen by the colors in the different layers near the fault. After a call to `processPartition`, the disconnected blocks have been reassigned a new index, and the number of partitions has increased from 60 to 72.



(a) Permeability setup

(b) Results for `partitionUI`



(c) Results for `partitionLayers`

(d) Results after `processPartition`

Figure 5.1: Outputs from the MRST coarse grid module

5.3 Dual grid

While a dual grid is trivial to define for simple grids such as the one shown in Figure 3.1, defining a dual grid for more complex structures is difficult. The intuitive definition of a dual grid,

Definition 5. *For some grid $\bar{\Omega}$ let the geometric center points of each block i be $\bar{x}_i$. The dual grid, $\tilde{\Omega}$, is defined by letting $\bar{x}_i$ from neighbouring coarse blocks be connected, and letting this partitioning of the domain represent a grid.*

Creating a fully realized dual grid using this definition for general unstructured

grids is hard. If the domain is not a convex set, a dual grid created by this definition is not always well defined:

Definition 6. *A set is convex if* $\forall$ *points* $p_i, p_j \in \Omega$ *if the points defined by*

$$(1-\theta)p_i + \theta p_j \tag{5.1}$$

$\forall \theta \in \mathbf{R}(0,1)$ *are also all found in* Ω.

This is apparent from Figure 5.2, where a non-convex domain of three coarse triangular elements give rise to a dual grid which exists outside of the domain (marked in blue). This is of course highly unwanted, as the permeability field may be undefined there and the dual element may cross over restrictive boundary conditions.

Fortunately, the operator formulation of the MsFV.method described in Chapter 3 only requires an ordering of nodes into central ($\mathcal{I}_n$), edge ($\mathcal{I}_e$) and inner ($\mathcal{I}_i$) nodes for the dual grid to work. Ordering each fine cell to a specific coarse block is only required for the primal grid, which simplfies the generation of both grids significantly:

We can use existing modules to generate the primal grid $\bar{\Omega}$, since generating coarse grids is an operation common to many multiscale and upscaling meth-

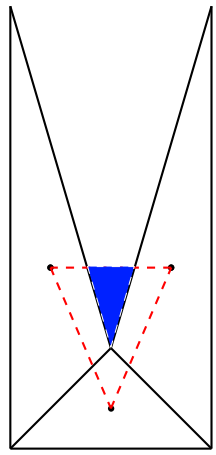


Figure 5.2: A straightforward interpretation of the dual grid can fail for non-convex domains

ods. From this we can use Definition 5 and subdivide each coarse block into different categories. There are some challenges to this approach:

1. How do we ensure that the edge cells across different $\bar{\Omega}_i$ are connected over the boundary? If this fails, the blocks of system matrix will not be disconnected and the solver will solve an almost full system.

2. How can the solver handle non-neighbouring connections where the logical neighbour is not the physical neighbour, for instance when the problem has faults?

Several approaches were considered for the generation of grids. For 2D, the grid generation is simple: We can for example use a shortest path algorithm on the graph of the system matrix A from $\bar{\Omega}_i$s center to its face centroids. If we choose a simple unit cost for each node traversal or weight it according to the corresponding cell permeability, we will get the correct grid for unstructured 2D cells. In 3D, however, this fails because the generalization of a general line is a *surface* and this has an geometric interpretation which is hard to implement for graph algorithms. Ensuring that the dual blocks $\tilde{\Omega}_i$ are closed,

Definition 7. *A dual block $\tilde{\Omega}_i$ will be defined as closed iff no fine cell $\Omega_k \in \tilde{\Omega}_i$ categorized as an inner cell $\mathcal{I}_i$ is connected through the system matrix A directly to a $\Omega_j \in \tilde{\Omega}_{h \neq i}$ also categorized $\in \mathcal{I}_i$.*

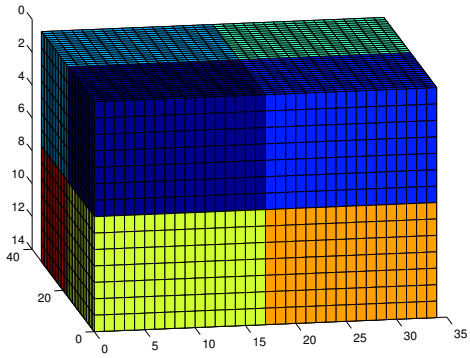
is a big challenge. Since unstructured grids can have a potentially very high number of faces in special cases, this is not necessarily always achievable. When starting the work on the possible partitioning schemes, several requirements were designed:

1. The scheme should partition simple right angled Cartesian grids correctly. This is the simplest goal, but it is still important as any scheme which fails on Cartesian grids are bound to fail on more complex grids.

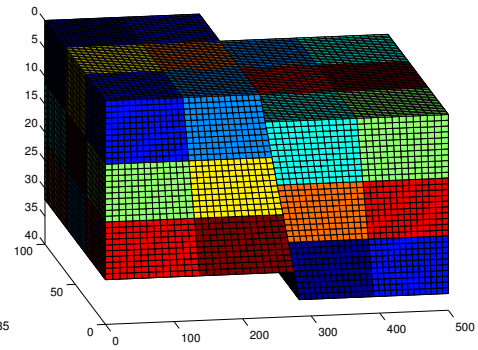
2. The scheme should handle non-neighbouring connections

3. The scheme should have low computational complexity to ensure fast grid generation

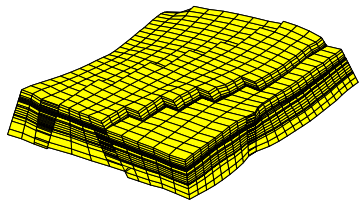
4. All the required data should either be easy to compute or already exist in MRST routines for geometry



(a) Simple cartesian grid



(b) Angled fault grid



(c) A coarse block subset of real geometry

Figure 5.3: Test grids for partitioning schemes with the coarse blocks in different colors.

5.3.1 Planar partitioning scheme

Since the MsFV-method using the operator formulation only considers geometry during the generation of the coarse grids, it was important to construct a scheme which would correctly partition simple Cartesian grids to begin the implementation. If the solver is implemented using encapsulation so that the partitioner is independent from the rest of the solver, it is easy to test out new algorithms without changing the rest of the solver.

The planar scheme is based on the assumption of each coarse block $\bar{\Omega}_i$ will be close to a right angled hexahedron. The scheme will still produce results for

other types of coarse blocks, with varying degrees of accuracy.

The algorithm picks a *pivot point* which is some corner of $\bar{\Omega}_i$. In MRST the coarse grid is not a fully defined grid with corners, but is rather defined by the fine cells which spans the block. To find a pivot point, we simply extract all the points corresponding to corners of fine cells on each surface of $\bar{\Omega}_i$. We then select an arbitrary surface and find the set difference with the points from other surfaces until we are left with a single point and three surfaces which are each other's neighbours in 3D space. While we are theoretically guaranteed to always find such a point when working with true 3D grids, occasionally special cases generated by MRST can fail this selection process. The algorithm detects this, and restarts with a new coarse face if the process fails to find a pivot after checking all neighbouring surfaces for a pivot point.

To ensure that the volume of each $\tilde{\Omega}_i$ will be of the approximate same size as each $\bar{\Omega}_i$ and that the edge nodes will be connected to edge nodes from the neighbouring $\tilde{\Omega}_i$, the centroids of the coarse faces will be used in the partitioning scheme. The three coarse neighbour face centroids define, pairwise along with the coarse block volume centroid, a plane which divides a cube in two parts of equal volume.

We will use a particular definition of a plane for this purpose,

Definition 8. *For a vector normal to the plane* $\mathbf{N}$ *and a single point* $\mathbf{p}_0$ *on the plane, the plane itself is defined as the set of all points* $\mathbf{p}$ *for which*

$$\mathbf{N} \cdot (\mathbf{p} - \mathbf{p}_0) = 0. \tag{5.2}$$

While this form can verify if a point is on the plane, it also has an additional benefit: The sign of (5.2) when evaluated for points not on the plane will signify which side of the plane a point is situated in. This is apparent if we consider the definition as a dot product: The sign of the solution will depend on the angle between the vector from the plane point to the point under consideration and the normal vector because

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos\theta. \tag{5.3}$$

This leads to the points being on the plane when this angle is $\pi/2$. Since $\cos(\pi/2 + dx)$ is negative and $\cos(\pi/2 - dx)$ is positive for small perturbations dx, taking

$$s(\mathbf{p}) = \frac{\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0)}{|\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0)|} = \operatorname{sign}(\mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0)). \tag{5.4}$$

on a set of points will classify their position relative to the plane.

The edge cells $\Omega_h \in \mathcal{I}_e$ will be any cells intersected by these planes. Since each cell is comprised of points we can identify the fine cells Ω_k inside $\bar{\Omega}_i$ intersected by each plane using (5.4) on the set of points $\mathbf{p}_j \in \Omega_k$:

$$\text{Category of } \Omega_k = \begin{cases} \mathcal{I}_n & \text{if the centroid of } \bar{\Omega}_i \text{ is inside } \Omega_k \\ \mathcal{I}_e & \text{when } \left|\sum_{\mathbf{p}_k \in \Omega_k} s(\mathbf{p}_k)\right| < N \\ \mathcal{I}_i & \text{otherwise} \end{cases} \tag{5.5}$$

Where N is the number of points defining the cell [1]. The sum for $\mathcal{I}_e$ gives the correct answer because $|s(\mathbf{p})| = 1 \forall \mathbf{p}$ so that the absolute value of the sum will equal the number of points in the cell if all points are on the same side of the plane. In Figure 5.4b this process is shown in the xy-plane along with the resulting categories of the fine cells.

This scheme obviously work for hexahedral coarse blocks in Cartesian grids: Since the cubes have equal faces, any choice of pivot will result in three faces with orthogonal normal vectors. This result of the algorithm on a single coarse block is shown in Figure 5.5a, with the three face centroids shown in red and the block centroid shown in green. When several of these blocks are assembled together as shown in Figure 5.5b, the centroids of the faces coincide across the coarse blocks, and the dual grid can be seen intersecting the different primal blocks (in different colours).

However, there are some obvious problems with this scheme: The pivot point is arbitrary and for more complex geometrical structures, the scheme can fail to partition the domain correctly. Potentially, for a special case primal grid block, the pivot can be a point where three almost parallel faces meet and the partitioning scheme will generate three planes which are very similar. There is also the issue of centroids only matching on three faces - while we know that the faces will start at a suitable point, there are no guarantees that they will end up somewhere reasonable.

Cartesian block

The Cartesian test case from Figure 5.3a is shown in Figure 5.6a. As expected, the results are good: The planes connect across edges of primal blocks and a

[1] This also works mathematically if we define the sum over *all* points inside the fine cell, but this is impossible to actually implement.

right angled primal grid gives a right angled dual grid. The only problems observed for this test case is that with certain grid sizes the scheme ends up creating faces of coarse blocks two blocks thick. However, as seen in Section 7.4, this is not much of a problem in terms of solution quality and is pretty much unavoidable for more complex grids.

Sloped fault

The sloped fault, which was shown in Figure 5.3b, has unsurprisingly correct partitioning for the right angled blocks. However, near the fault, the sloped edges cause problems for the algorithm and the scheme fails as shown in Figure 5.6b. Why does the planar scheme fail near the fault? The answer lies in the definition of the primal grid in MRST: Along the fault, each coarse block has more than six faces to ensure that each face only touches two coarse blocks. The algorithm then chooses pivot points without being able to distinguish between the faces and the planes along the xy-plane then have seemingly arbitrary centroids as starting points. Because the grid is no longer right angled at all faces, the planes nearly parallel to the xy-plane end up not connecting at any primal faces at the fault.

This is a situation typical of real geological models where it is impossible to create a right angled coarse hexahedron and still preserve the complex underlying geometry. Having small coarse faces on the edges where the surface is non-smooth is to be expected, and should be handled by the dual grid scheme.

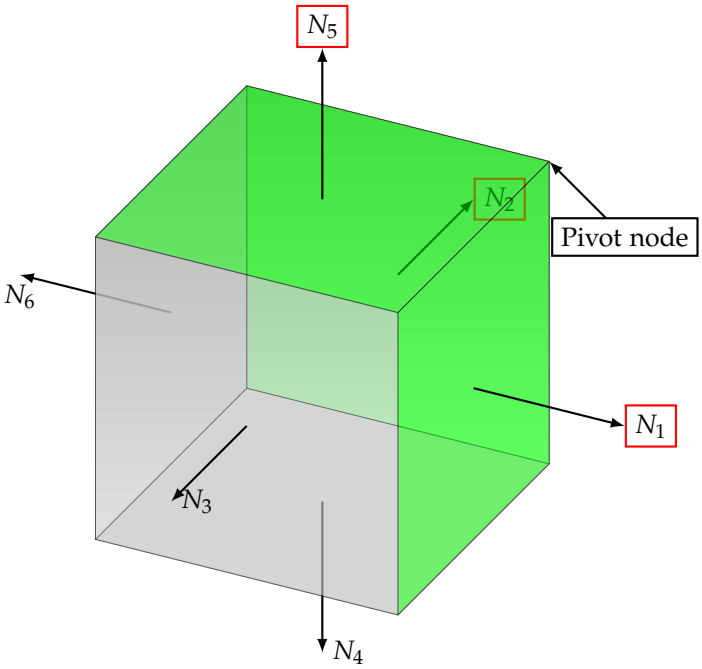
Realistic subset

In the realistic reservoir model subset originally shown in Figure 5.3c, the planar scheme produces very bad results (Figure 5.6c). The pivot choice is unfortunate and results in the three nearly parallel coarse faces which then leads to three partitioning planes intersecting the same fine cells. The results of such failures can be disastrous, leading to anything from reduced solution accuracy to singular systems.

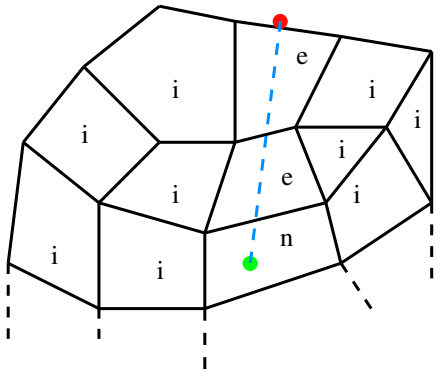
Conclusion

To summarize the planar algorithm, it is important to note that this is a first attempt towards generating dual grids. It is a failure for typical unstructured

grids, but handles the basic Cartesian grid very well. This was significant because it enabled the development and testing of the rest of the MsFV-method on regular grids before revisiting the partitioning problem later. Since the partitioning scheme is the only part of the module which requires geometrical handling, it is easy to swap in another scheme if the code is well structured and developed with encapsulation in mind.

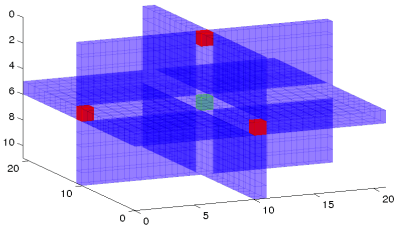


(a) A typical coarse block

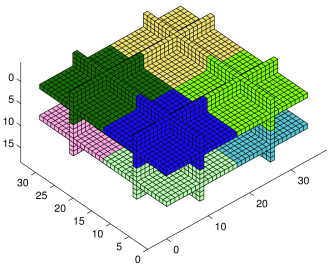


(b) The plane categorizes fine cells

Figure 5.4: (a) shows a typical coarse block along with the pivot node and the three faces selected by the planar partitioning algorithm. (b) shows how, in an 2D cutout, the planar algorithm will categorize fine cells.

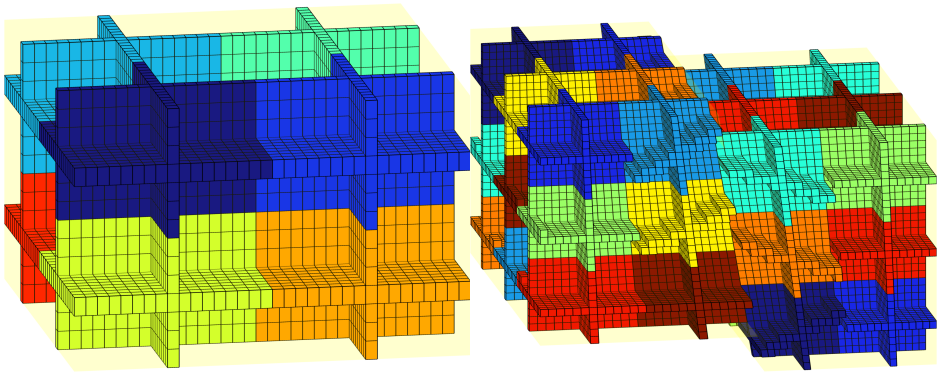


(a) Planar partitioning on a single coarse block



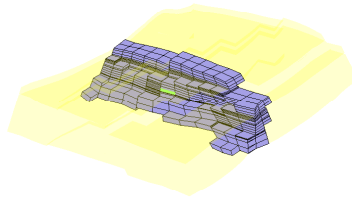
(b) Planar partitioning on four coarse blocks

Figure 5.5: The planar algorithm on simple right angled Cartesian grids



(a) Simple cartesian grid

(b) Angled fault grid



(c) A coarse block subset of real geometry

Figure 5.6: Results for the planar algorithm

5.3.2 Improved planar partitioning

Because the planar partitioning scheme had problems handling discontinuous features typical of realistic grid models, another scheme was created which sought to ensure that the planes would be connected across all main faces of the primal block $\bar{\Omega}_i$. The assumption of the algorithm is that each coarse block has, when visually inspected, roughly six main faces. The algorithm should be able to identify these main faces and ignore small faces which are close to each other.

Improved selection algorithm

The first step of the new algorithm is the *Improved selection algorithm*:

for all Blocks $\tilde{\Omega}_i \in \tilde{\Omega}$ **do**

Set normal vectors to be oriented outwards from $\tilde{\Omega}_i$

Find face T_1 (Top) and T_2 (Bottom) using maximum and minimum of the face normal vectors' z-components respectively

Sort the remaining faces by area and select the four largest for further processing

Select one of the four faces arbitrarily and denote it A_1. Find A_2 by

$$\max(\arccos(N_{A_1}^{xy} \cdot N_{F_i}^{xy}))$$

where $N_{F_i}^{xy}$ are the x & y components of the normal vectors of the three remaining faces.

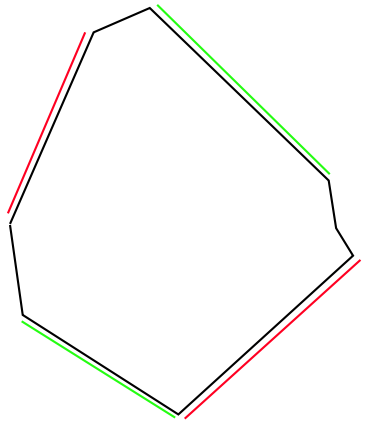
The remaining two faces are B_1 and B_2.

end for

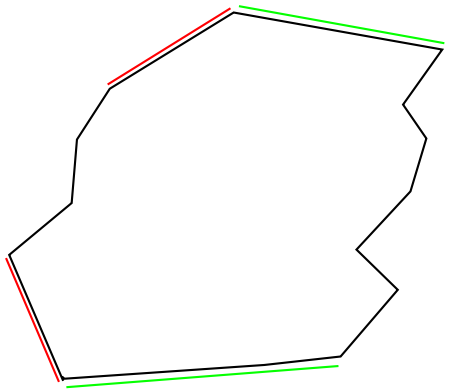
This algorithm finds six faces of $\tilde{\Omega}_i$ and matches them up so that they are hopefully opposing each other. By selecting the minimum and maximum of the normal vector z-components for the top and bottom (the normal vectors are unitary, $\|N\| = 1$) we exploit the fact that most realistic grids seem to have well defined top and bottom faces.

However, the same approach is not successful for the other four faces: We cannot know of the coarse faces will be aligned with any axes. Therefore we simply pick the four largest faces (to ensure that we do not pick degenerate faces) and maximise the angle in the xy-plane between them using arccos on the dot product of the $N_{F_i}^{xy}$ vectors. This approach will identify the correct faces if the coarse partitioning scheme results in approximately Cartesian blocks.

Two examples illustrate this: Since the algorithm chooses four faces weighted by area before maximizing the angle between them, the algorithm will work for some coarse blocks, exemplified by Figure 5.7a where the pairs (red and green) are selected correctly. The same step applied to a coarse block with two faces comprised almost entirely of smaller faces will result in a bad choice of coarse faces as seen in Figure 5.7b.



(a) Selection algorithm succeeds because the largest faces are opposing each other



(b) Selection algorithm fails because the four largest faces are not opposing each other.

Figure 5.7: A 2D cut-out showing coarse blocks for which the improved selection algorithm selects correct faces (5.7a) and suboptimal faces (5.7b). Each pair of faces selected are illustrated with red and green respectively.

Using several planes to connect the grid

Once we have these points it is obvious that like the original planar partitioning scheme, we can construct planes by selecting two sets of faces from T, A, B, taking one face centroid from each set and using it along with the block volume

centroid C. Any such combinations can be used to create a plane which touches the center and two face centroids.

We desire a function which, for two sets of faces $\{A_1, A_2\} = A$ and $\{B_1, B_2\} = B$[2] and the block centroid $\mathbf{C}$, constructs a surface such that the surface function changes sign at all the face centroids $\mathbf{C_{A_i}}$ and $\mathbf{C_{B_i}}$ as well as at $\mathbf{C}$. In addition we want this surface to be continuous to ensure that the partitioning results in a closed surface (Definition 7). It is also preferable if the surface is at least piecewise smooth to avoid clustering of fine cells when determining which fine cells are intersected.

The solution is to use a special *Divider plane* to further partition $\tilde{\Omega}_i$. This plane is constructed as to ensure that it *only* intersects $\mathbf{C}$ and not any other face centroids. To achieve this, a normal vector is constructed so that it lies between the vectors from $\mathbf{C}$ to $\mathbf{C_{A_1}}$ and $\mathbf{C}$ to $\mathbf{C_{B_1}}$:

$$N_{Divider} = \frac{\mathbf{C_{A_1}} - \mathbf{C} + \mathbf{C_{B_1}} - \mathbf{C}}{2} = \frac{\mathbf{C_{A_1}} + \mathbf{C_{B_1}}}{2} - \mathbf{C} \tag{5.6}$$

$N_{Divider}$ along with $\mathbf{C}$ and (5.2) then defines a plane ($P_{Divider}$) which partitions $\tilde{\Omega}_i$ in two halves. This is shown in Figure 5.8a. Using this plane, we can define

$$\delta_{Divider}(\mathbf{p}) = \begin{cases} 1 & \text{if } P_{Divider}(\mathbf{p}) \text{ is equal to } P_{Divider}(\mathbf{C_{A_1}}) \\ 0 & \text{otherwise} \end{cases} \tag{5.7}$$

and

$$\bar{\delta}_{Divider}(\mathbf{p}) = (\delta_{Divider}(\mathbf{p}) + 1) \mod 2. \tag{5.8}$$

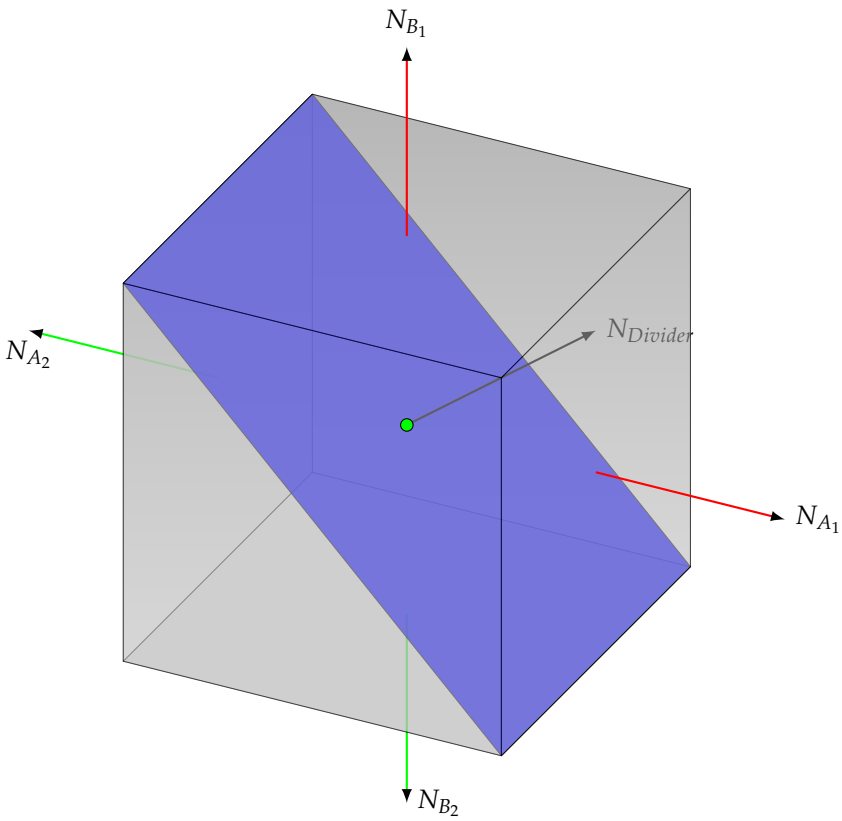
($\bar{\delta}_{Divider}$ is obviously 1 where $\delta_{Divider}$ is 0 and vice versa)

We now want to create a combination of the planes defined by $\mathbf{C}, \mathbf{C_{A_1}}, \mathbf{C_{B_1}}$ denoted P_1 and $\mathbf{C}, \mathbf{C_{A_2}}, \mathbf{C_{B_2}}$ denoted as P_2, which is zero at all five points:

$$P_{improved}(\mathbf{p}) = P_1(\mathbf{p})\delta_{Divider}(\mathbf{p}) + P_2(\mathbf{p})\bar{\delta}_{Divider}(\mathbf{p}). \tag{5.9}$$

The resulting function will be zero at the centroids while still representing a continuous surface and can be used in the same manner as the earlier plane without modifying the implementation significantly.

[2] A and B can be either of T, A, B mentioned in the preceding paragraph



(a) The divider plane

Figure 5.8: Given two opposing pairs of faces A and B, the divider plane (blue) partitions the coarse cell into two halves. Each normal vector originates on the centroid of the corresponding faces.

Cartesian block

The result of the improved planar algorithm on the base Cartesian case can be seen in Figure 5.9a. The result is the same as the original plane scheme: Since for the opposing faces we have $N_{A_1} = -N_{A_2}$, both $P1, P2$ are equal and the algorithm is simply a more expensive way to get the same result. It is worth noting that if checking the position of a point relative to a single plane has a cost of k, this scheme has a cost of $3k$ - one evaluation for each plane and one

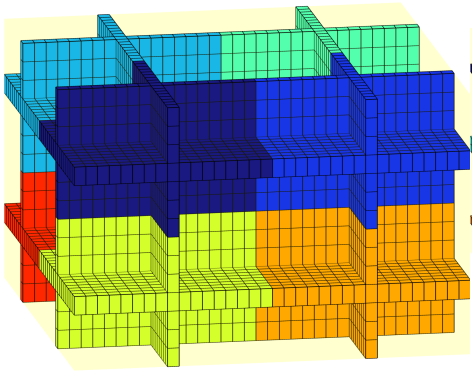
for the orientation delta. If the grid is right angled and Cartesian and speed is essential, the improved algorithm will result in a needless speed hit.

Sloped fault

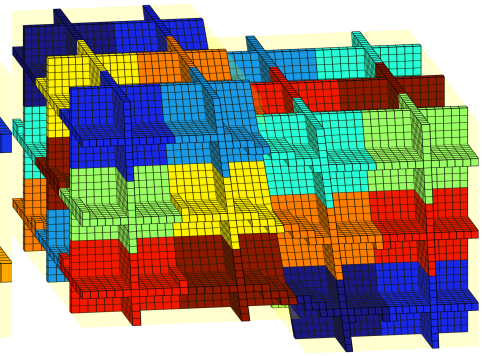
The results for the sloped fault are improved from the previous case: Unsurprisingly, the Cartesian blocks are partitioned correctly. In the blocks connected to the fault, the results are more interesting: The algorithm correctly identifies the significant faces and partitions across the fault just as desired.

Realistic subset

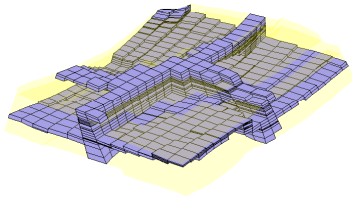
The biggest improvement is seen in Figure 5.9c: Instead of getting overlapping planes, the algorithm successfully finds a partitioning of the domain reminiscent of the cross structure seen in Cartesian blocks. Of course, the algorithm can still fail if the blocks does not have any large faces altogether or if the coarse block is not convex - the planes can then risk cutting through empty space. The algorithm is based on the assumption of the coarse block partitioner to be able to produce well formed blocks and as such manual tuning may be required. The advantage of this approach is that any progress made on coarse grids for other multiscale / upscaling methods will improve the MsFV-method results automatically.



(a) Simple cartesian grid



(b) Angled fault grid



(c) A coarse block subset of real geometry

Figure 5.9: Results for the improved planar algorithm

5.3.3 Polynomial partitioning

Another approach considered was to use polynomials to interpolate the points. This treatment will assume that the data available are the face centroids as well as the block centroids, which coincides with what MRST's coarse grid module produces. The selection algorithm from Section 5.3.2 is used to select the points, but instead of using each tuple A_1, A_2, B_1, B_2, C to create two planes, a single interpolation polynomial is created.

This is done using a Vandermonde matrix. While creating an interpolation polynomial can be done in many ways, this is one of the easiest to implement with access to linear algebra subroutines and the results often coincide with Lagrangian interpolation. Since we have a set of five points p for each surface second degree polynomials is an obvious choice to get the required degrees of freedom.

We require a set of coefficients c_i such that

$$f(x, y, z) = c_0 + c_1 x + c_2 y + c_3 z + c_4 xy + c_5 xz + c_6 yz + c_7 x^2 + c_8 y^2 + c_9 z^2 \quad (5.10)$$

and

$$f(x_j, y_j, z_j) = 1 \forall (x_j, y_j, z_j) \in p \quad (5.11)$$

.

Since the coefficients are constants, this is a linear equation set:

$$V\mathbf{c} = \begin{bmatrix} 1 & x_1 & y_1 & z_1 & x_1y_1 & x_1z_1 & y_1z_1 & x_1^2 & y_1^2 & z_1^2 \\ 1 & x_2 & y_2 & z_2 & x_2y_2 & x_2z_2 & y_2z_2 & x_2^2 & y_2^2 & z_2^2 \\ & & & & & \vdots & & & & \\ 1 & x_n & y_n & z_n & x_ny_n & x_nz_n & y_nz_n & x_n^2 & y_n^2 & z_n^2 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad (5.12)$$

The polynomial is set to 1 at the interpolation points in order to force a nontrivial solution. To get the normal zero form polynomial required for categorization of points, the function evaluation is simply:

$$\tilde{f}(x, y, z) = f(x, y, z) - 1. \quad (5.13)$$

While a second degree polynomial obviously provides enough degrees of freedom for five points, there is no guarantee that the resulting surface will be appropriate for partitioning.

Cartesian block

In Figure 5.10a we can see that the polynomial scheme fails for simple Cartesian blocks. While the grid is connected across coarse blocks at the face centroids, the polynomial properties end up warping the surface near the edges where no points are provided. This results in none-closed dual cells and should be considered a failure. If more data on the coarse cell faces were provided, the results could probably be improved into something reminiscent of the planar algorithm.

Sloped fault

In an interesting twist, the polynomial scheme applied to the fault as shown in Figure 5.10b results in good results across the faultline, but some strictly Cartesian blocks fail, again because of the low amount of points. This shows that the polynomial scheme has some promise, but with the current available data the results are inferior to the improved planar partitioning algorithm.

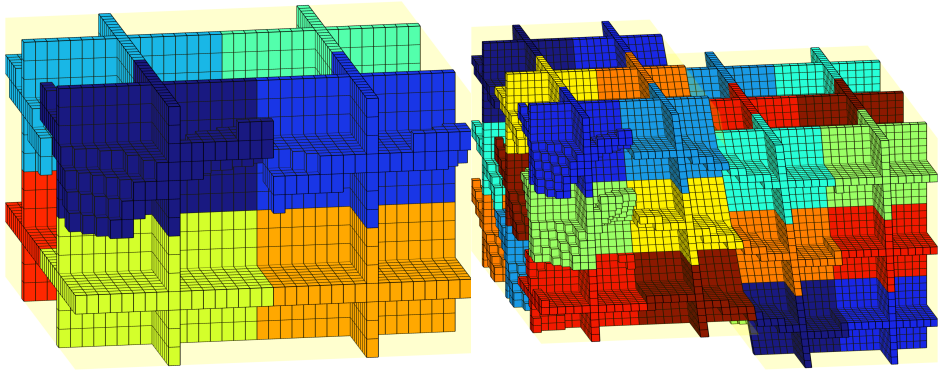
Realistic subset

For the realistic grid subset, the polynomial interpolation looks similar to the planar scheme (Figure 5.10c). However, the same warping effects are present and the scheme categorizes more fine cells as edge cells than the improved planar scheme - probably because of the curvature of the planes which ends up intersecting more fine cells than a single plane more or less aligned with the coordinate axes.

5.3.4 Face merging

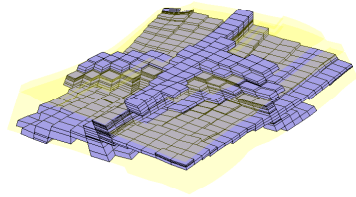
An alternative to the face selection algorithm in Section 5.3.2 was created during the testing process. The idea is to, when faced with a large amount of small faces on a coarse block, to merge smaller faces until a reasonable amount of faces exist. We will assume that the top and bottom face is preselected so that we require only four faces from the remaining set.

while $N_{faces} > 4$ **do**
  $F_{min} \leftarrow$ smallest face
  $F_{neighbour} \leftarrow$ find$(\arccos(V_{F_{min}} \cdot V_{F_i}) \forall F_i \in faces$



(a) Simple cartesian grid

(b) Angled fault grid



(c) A coarse block subset of real geometry

Figure 5.10: Results for the polynomial partitioning algorithm

$F^A_{merged} \leftarrow F^A_{min} + F^A_{neighbour}$
$F^C_{merged} \leftarrow F^A_{min} F^C_{min} + F^A_{neighbour} F^A_{centroid}$
Faces $\leftarrow$ Faces $- F_{min}, F_{neighbour} + F_{new}$
end while

Where superscript A refers to area, C the centroid and V_{F_i} the vector from C to F^C_i.

While this algorithm produces a better choice of faces for degenerate blocks with a high amount of faces, it will generally not connect the dual grid across primal boundaries because the probability of merging faces in the same way for different blocks is low. The approach combined with other selection heuristics

may have some merit in applying a partitioning scheme meant for cubic coarse blocks to for example PEBI-grids where a fitting of right angles will lead to a large amount of coarse faces. For the scope of this thesis, however, it should be regarded as a failure.

5.3.5 Logical partitioning schemes

While the partitioning schemes utilising geometric information seem quite successful at handling faults and some types of complex geometry, they have a weakness in that they are dependent on being able to identify coarse faces correctly. This is not always the case: A grid can be fully Cartesian when looking at the logical indices i, j, k for each cell, but highly challenging when visualized in space. As an example of this, the grid shown in Figure 5.11a will be used, where a regular 2D grid has had its coordinates permuted by a function,

$$f(x, y) = 0.1 \sin(\pi x) \sin(3(-\pi/2 + \pi y), \tag{5.14}$$

to get a grid which is logically Cartesian, but with large varieties in cell areas and faces. The resulting coarse grid has no obvious large faces which the algorithm can select for connecting the dual grid over centroids.

There is also the question of speed. A complex algorithm using planes requires a large amount of lookups in the data structures to find the points corresponding to fine cells, and a lot of time is spent categorizing every single point in relation to the different partition functions. For simpler grids without faults, such as the grid in Figure 5.3a, a lot of time could be saved by a faster algorithm.

To partition using a logical algorithm, the we require the logical coordinates of each fine cell. If the grid is ordered in x and y direction with N_x and N_y nodes respectively, the logical indices i, j can be extracted from the index α by the way of

$$i = mod(\alpha, N_x) \tag{5.15}$$

$$j = mod(\alpha - i, N_x N_y). \tag{5.16}$$

This relies on the fact that the nodes are ordered after each other, and is done in MATLAB using the `sub2ind` function.

To partition the domain so it becomes the correct dual grid for our coarse grid, we have to partition the domain in the same manner as the `partitionUI` routine. `partitionUI` uses a load balanced linear distribution to distribute the

logical indices in each direction across coarse blocks and we will do the same. Because of time constraints, we will limit ourselves to grids where the inactive blocks are on the edges of the domain.

The algorithm finds the minimum and maximum coordinate in each logical direction, which is then used to construct a load balanced linear range across that logical direction. The center points of each interval is then selected, because we know that this is where the edges of the dual grid will intersect. Once all such points are found, they can then be used to categorize all cells with the same logical index in any dimension as edge nodes. These are stored as logical arrays indexing into the list of cells.

To find the inner cells and central cells, it is then just a manner of doing intersection operations on the logical vectors: The central cells are obviously the cells categorized as belonging to all edges and the inner are those belonging to no edges at all. When implemented using vector operations to do most the heavy lifting, the code looks like this:

```
function dual = partitionUIdual(CG, blockSizes)
G = CG.parent;
n = G.cells.num;
% Decrement all positions because working with zero indexing is
    easier
% to work with when partitioning...
cells = G.cells.indexMap-1;
% Find positions of all the nodes in ijk space
spaces = cell(G.griddim,1);
[spaces{:}] = ind2sub(G.cartDims, G.cells.indexMap);
uniques = cell(G.griddim,1);
for d = 1:G.griddim
    ms = min(spaces{d});
    Ms = max(spaces{d});
    % Find the number of positions in the current dimension
    M = Ms - ms + 1;
    % Do a load balanced distribution of the positions in the same
        manner
    % as in the primal grid. Cast to double to avoid integer
        division
    % problems.
    balanced = lbLinDist(double(0:(Ms-ms)), double(M), double(
        blockSizes(d)));
    u = unique(balanced);
    uniques{d} = zeros(numel(u),1);
    index = 1;
    for i = 1:numel(u)
        % Keep a running index and find the midpoints of the
            intervals
```

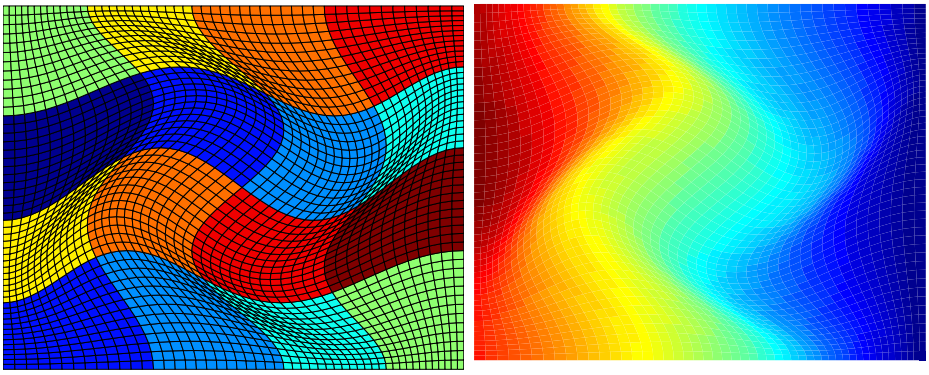
```
        % to use as centerpoints for the dual grid
        Ni = sum(balanced == u(i));
        uniques{d}(i) = index + round(Ni/2) - 1;
        index = index + Ni;
    end
end
% Gather all the different positions in each D where the grid has dual edges
tmp = zeros(n,G.griddim);
for d = 1:G.griddim
    ii = uniques{d};
    for i = 1:numel(ii)
        tmp(:,d) = tmp(:,d) | spaces{d} == ii(i);
    end
end
% The central nodes are those which are in two sets of edges in 2D, and 3
% sets of edges in 3D. Increment to get one indexing.
dual.nn =  cells(sum(tmp,2)==G.griddim)+1;
% Any node which exists in more than one edge must be filtered to
% decouple the edge system. Increment to get original indexing.
dual.lineedge = cells(sum(tmp,2)>1)+1';
% Do an OR on each dimension to find all nodes corresponding to the
% midpoints of each primal coarse block
if G.griddim == 3
    tmp = tmp(:,1) | tmp(:,2) | tmp(:,3);
else
    tmp = tmp(:,1) | tmp(:,2);
end
% Find the indices of nodes on the edge, increment to get back to one indexing
dual.ee = cells(tmp == 1)+1;
% Use the new behavior of setdiff to ensure forward compatability
% Ensure that the different edges are distinct
dual.lineedge = setdiff(dual.lineedge, dual.nn,'R2012a');
dual.ee = setdiff(setdiff(dual.ee, dual.lineedge,'R2012a'), dual.nn, 'R2012a');
dual.ii = setdiff(G.cells.indexMap, [dual.ee; dual.nn; dual.lineedge]);
```

The coordinates are never used when constructing the dual grid, resulting in an algorithm which is invariant to transformations on the underlying coordinates. When applied to the grid in Figure 5.11a, the planar algorithm fails to connect the dual grid across dual edges (Figure 5.11e). Logical partitioning, however, is successful in creating a dual grid in spite of the transformed coordinates, shown in Figure 5.11f. When applied to a simple flow channel boundary condition and lognormal permeability (see Section 7.3.1 for more details), the solution using the logical algorithm (Figure 5.11d) is markedly better than the planar

algorithm (Figure 5.11c) when compared to the TPFA reference solution (Figure 5.11b).

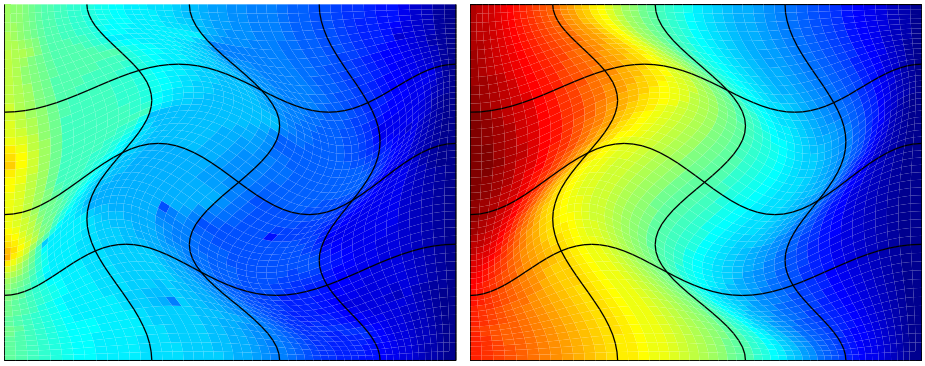
The numerical error verifies what should be obvious from the figures: The relative norm of the error is much lower for the logical case ($\|e\| = 0.012$) than for the planar case ($\|e\| = 0.454$).

When it comes to the test cases, the logical algorithm can only be applied to the structured grids. The results for the Cartesian case is, as expected, good. The algorithm coincides with the planar algorithm in Figure 5.12a. For the fault, shown in Figure 5.12b, the algorithm fails to connect across the fault, as expected.



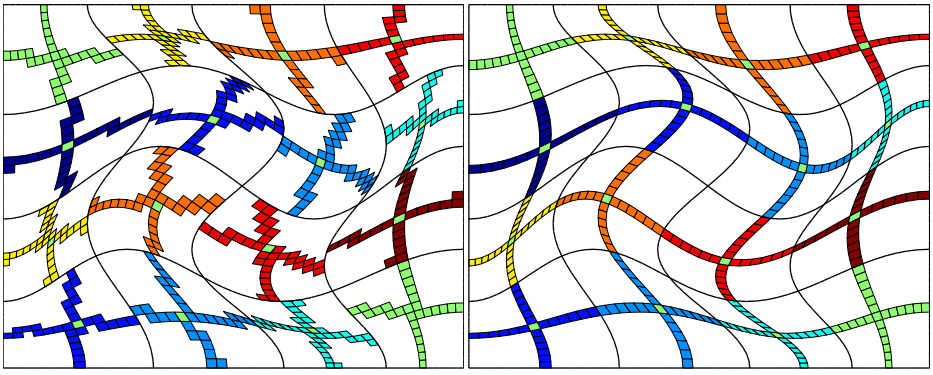
(a) The twisted grid

(b) TPFA Reference solution on the twisted grid



(c) MsFV-solution (Planar algorithm)

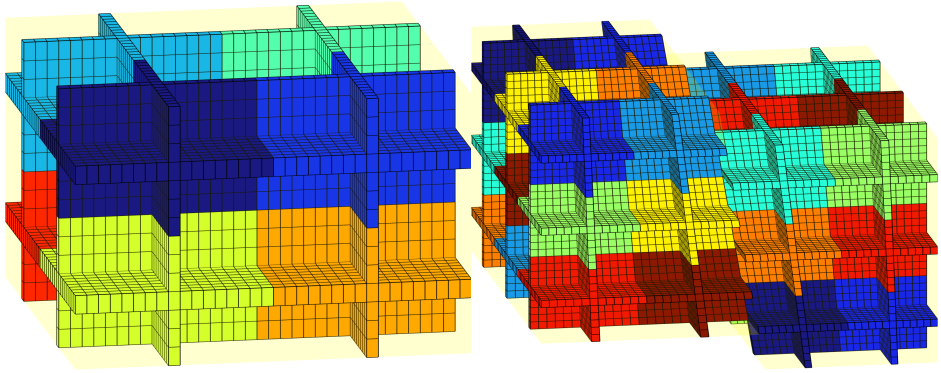
(d) MsFV-solution (Logical algorithm)



(e) Dual grid (Planar algorithm)

(f) Dual grid (Logical algorithm)

Figure 5.11: For a grid with twisted geometry, the logical algorithm improves the results greatly



(a) Simple cartesian grid (b) Angled fault grid

Figure 5.12: Results for the logical partitioning algorithm. Unstructured grid omitted because of missing Cartesian mapping.

5.3.6 Performance and implementation

The initial version of the dual grid module suffered from bad performance. This came from the calling of the anonymous categorization function for each fine cell, leading to a huge overhead from the many repmat-calls needed for this logic. Fortunately, categorizing all points in a single pass and then looking up the values for each fine cell was feasible, and the partition speed increased drastically. It is important to note that while speed is important for all parts of the MsFV-method, the partitioning step can be considered as a part of grid processing and, if stored, need only be done once for all manner of boundary conditions and permeability set ups. It is therefore not as critical as the pressure solution steps themselves.

The resulting code for a single coarse block looks like this:

```
1 nodeInd = cell(numel(blockInd),1);
2 nodeIndices = [];
3 current = 1;
4 % BlockInd is the indices of all fine cells corresponding to the
      current coarse block
5 for c = 1:numel(blockInd)
6     fa = cg.parent.cells.faces(cg.parent.cells.facePos(blockInd(c)):
          cg.parent.cells.facePos(blockInd(c)+1)-1);
7     tmp = [];
8     for f = 1:numel(fa)
```

```
9           tmp = [tmp cg.parent.faces.nodes(cg.parent.faces.nodePos(fa(
                f)):(cg.parent.faces.nodePos(fa(f)+1)-1))'];
10      end
11      nodeInd{c} = current:current+length(tmp)-1;
12      nodeIndices = [nodeIndices tmp];
13      current = current + length(tmp);
14  end
15  coords =  cg.parent.nodes.coords(nodeIndices,:);
16  orient = zeros(size(coords,1),cg.griddim);
17  for pp = 1:cg.griddim
18      % Save the orientation of all points
19      % func is a cell array of functions created by the partition
            scheme
20     orient(:,pp) = func{pp}(coords);
21  end
22
23  for c = 1:numel(blockInd)
24      nodePos = nodeInd{c};
25      for pp = 1:cg.griddim
26          orientation = orient(nodePos, pp);
27          if abs(sum(sign(orientation))) < numel(orientation)
28              % Categorize fine node as edge node
29              dual.edges{i,pp} =  [dual.edges{i,pp} blockInd(c)];
30              % If some function categorizes the cell as edge, we can
                    stop checking
31              break;
32          end
33      end
34  end
```

By using a cell array of anonymous functions for the partitioning scheme, the logic for categorizing points is separate from function generation and it is easy to implement new methods in the future.

While the different partitioning schemes have different complexity, the geometrical algorithms which uses planes to partition the domain should be similar in run time because the overhead of traversing data structures in MATLAB is large. The logical algorithm is a special case which should be drastically faster, but is not applicable to complex geometries.

A series of performance tests on the algorithms were run on a 2.67 GHz Intel Xeon CPU. Two different scenarios were considered for increasing domain sizes: Constant block size, where each coarse block was $5 \times 5 \times 5$ and constant block count, which always produces 5 dual blocks along each axis. The domain was $N \times N \times N$ with varying as $N = 50, 60, ..., 200$ giving between 125 000 and 8 000 000 nodes. The results can be seen in Figure 5.13. As is to be expected, the time required is mostly dependent on the number of coarse blocks, with

both planar and improved planar (listed as polyplane) using roughly the same amount of time. Constant block count is a lot faster, as is to be expected since the face selection and function generation happens only once for each coarse block. The logical algorithm is much faster, with a seemingly constant partitioning time near one second for all grid sizes.

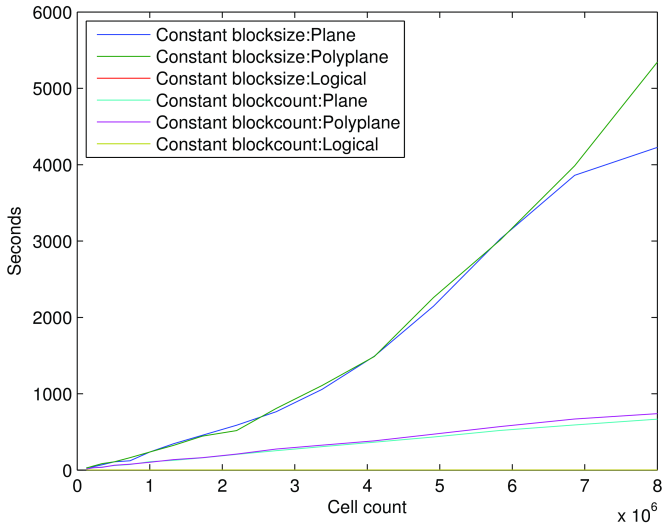


Figure 5.13: The performance of the different partition algorithms

5.3.7 Summary and further work

While a lot of progress has been made towards generalizing a dual grid, the schemes fail for some special cases. More complex partitioning schemes could be considered, for example by using splines instead of planes which curve along the geological layers of the model. A non-geometrical approach might also be successful, using search algorithms on the system graph or using more complex optimization principles to find surfaces which both ensure a closed surface and compensate for geological patterns.

Solving this problem for general grids is a difficult problem and requires the developer to strike a balance between execution speed, code complexity and geometrical flexibility. The current schemes developed in this thesis handle

complex unstructured grids from real data with a fairly high success rate and will automatically take advantages of improvements done in MRST's coarse grid module. Some degree of visual verification is required before use and for especially difficult situations, manual tweaking is required. The difficulty posed by imposing grids on unstructured grids is significant and while methods may be trivial to implement for purely academic cases, creating a general "one size fits all"-solution for unstructured grids with discontinuities is difficult and time consuming.

5.4 Permuting the system and handling wells

The first step of the algorithm is to create permutation matrices. For this step, we will assume that both a primal and a dual grid exists as defined in Section 5.3: One list for each category (inner, node and edge) containing the indices of the fine cells contained in each category of $\tilde{\Omega}$. We will also assume a partition vector for $\bar{\Omega}$ with one entry However, before creating the permutation matrices, we need to preprocess these vectors to handle the different types of wells supported in MRST. For the implementation, we require all rate wells to be defined *before* the BHP wells in the well data structure. The reasons for this will become apparent in the section on rate wells.

There are two types of wells supported in MRST: Rate wells and bottom hole pressure wells (BHP).

Bottom hole pressure wells

BHP wells are analogous to Dirichlet boundary conditions - a BHP well forces any connected cells to have a specific pressure value. In applied terms this could be used to model say a well kept at constant pressure while producing or injecting, depending on the pressure differential. The most common model for BHP wells is the Peaceman Well Model[Pea83] which makes it possible to calculate an equivalent source term from a vertical well with a given radius r_w. For a cell with widths h_x, h_y, pressure p_0, well pressure p_{wf} and a height h this source term is given as

$$q = \frac{2\pi(p_0 - p_{wf})\sqrt{K_x K_y}h}{\mu \ln(r_0/r_w)}, \tag{5.17}$$

where the equivalent well block radius given as,

$$r_0 = 0.28 \frac{\left[(K_y/K_x)^{1/2} h_x^2 + (K_x/K_y)^{1/2} h_y^2 \right]^{1/2}}{(K_y/K_x)^{1/4} + (K_x/K_y)^{1/4}} \tag{5.18}$$

for an anisotropic medium. We will not delve into the derivations behind this equation, but simply note that it reduces the well implementation to adding a source term in the right hand side of the linear system.

MRST adds an additional equation for this system, but since this equation is simply added to be able to be extracted later by the TPFA solver for the correct pressure, it can be omitted from the system and safely ignored, as long as we remember to insert the value in the solution state.

Rate wells

A rate well is analogous to a Neumann boundary condition - a rate well gives restrictions on the derivative in the point, representing a steady stream into or from the reservoir at that well. For rate wells, the implementation is slightly more complex than BHP wells: Another equation is added for each well, which is then connected to the appropriate well cells. The additional equation is a a discrete derivative restriction connected to some cell from the well cells, and at the same time the corresponding cells are connected with free flow conditions inbetween them according to the Peaceman well index.

To implement rate wells in MsFV-method, we cannot ignore the additional equations. After some testing, it was decided that the additional equations should be categorized as actual nodes would. This will ensure that the rate wells will be solved along with the equation system for the corresponding fine nodes. To do this, we must categorize the extra equations in the dual grid the same as the cells it corresponds to. A rate well containing more than one cell will be categorized in the order Node, Edge and Inner, because this is the importance of the nodes when it comes to influence the solution: Nodes are solved first using the coarse system, followed by interpolation to the edge and inner nodes.

The resulting code looks like this, before creating the permutation matrices:

```
% Number of cells in the system
Nc = N - Nw;
% "Real" nodes are all nodes not corresponding to BHP well cells
real_nodes = N - Nbhp;
```

```
5
6  passed = false;
7  for i = 1:Nw
8      w = opt.wells(i);
9      if strcmp(w.type,'rate')
10         if passed
11             error('Rate wells should come before BHP wells in the
                   msfvm solver')
12         end
13         contact_ii = intersect(w.cells, dual.ii);
14         contact_nn = intersect(w.cells, dual.nn);
15         contact_ee = intersect(w.cells, dual.ee);
16         if sum([any(contact_ii) ...
17                 any(contact_nn) ...
18                 any(contact_ee)]) > 1
19             warning('well:msfvmwellisect','Rate well intersects
                   several different node categories..')
20         end
21         if any(contact_nn)
22             dual.nn = [dual.nn Nc+i];
23         elseif any(contact_ee)
24             dual.ee = [dual.ee Nc+i];
25         else
26             dual.ii = [dual.ii Nc+i];
27         end
28         %add the well nodes to the same partition as the connected
29         %cells (the first cell is chosen arbitrarily
30         CG.partition = vertcat(CG.partition, CG.partition(w.cells(1)
               ));
31     else
32         passed = true;
33     end
34 end
```

Note that we also update the partition vector, to ensure that the operators which require an ordering in $\bar{\Omega}$ still work (notably the restrict operator χ and the permutation operator for the flux reconstruction $\bar{P}$).

We require BHP wells to be ordered after Rate wells so that the `dual_partition` function does not have side effects (reordering the well structure) or become too complex.

Once all nodes are accounted for and categorized, creating permutation matrices is trivial. The permutation matrix for $\tilde{\Omega}$ is just a straightforward concatenation of the different categorizations which is then used as indices for each row in the permutation matrix:

```
1      ordering = double([dual.ii dual.ee dual.nn]);
2      dual.P = sparse(1:Nc,...
```

```
                    ordering,...
                    1) > 0;
```

This results in one nonzero element for each row in the permutation matrix and the greater than sign transforms it into a logical sparse matrix for efficient storage.

The creation of the permutation matrix for $\bar{\Omega}$ is created in a similar way, using find to ensure that all indices from the same coarse block ends up in sequence:

```
   ind = 1;
   for i=1:CG.cells.num
       tmp = find(CG.partition == i);
       ordering(ind:(ind+length(tmp)-1)) = tmp;
       ind = ind + length(tmp);
   end
   dual.P_flux = sparse(1:real_nodes,...
                    ordering(1:real_nodes),...
                    1) > 0;
```

5.5 Implementing the operators

While some of the operators described in Chapter 3 are trivial to implement, there are some pitfalls when implementing the operator formulation of the MsFV-method.

First, we need the different blocks described in the operator treatment. This is done by permuting the system and slicing the correct values:

```
n_i = length(DG.ii); n_e = n_i + length(DG.ee); n_n = n_e + length(
    DG.nn);
Nf = DG.N;
% Permute the system and ignore BHP wells
A = DG.P*state.A(1:Nf, 1:Nf)*DG.P';
r = DG.P*state.rhs(1:Nf);
% Internal nodes' influence on internal nodes
A_ii = A(1:n_i, 1:n_i);
% Edge nodes' influence on internal nodes
A_ie = A(1:n_i, (n_i+1):n_e);
% Edge nodes' influence on edge nodes
A_ee = A((n_i+1):n_e, (n_i+1):n_e);
% Center nodes' influence on edge nodes
A_en = A((n_i+1):n_e, (n_e+1):n_n);
```

One important thing to note is that while the operator formulation is formulated so that explicit inverses seem necessary, this can lead to extremely inefficient implementations. While M_{ee}^{-1} and A_{ii}^{-1} are needed for many operators, forming them explicitly is not advised. In MATLAB,

```
inv(A)*B; % Explicit inverse
A\ B;        % Equivalent linear solver formulation
```

have the same result because $Ax = b$ can be solved as $x = A^{-1}b$ which is solved using the backslash/mldivide operator in best practice MATLAB. Although we need to do this several times to form all operators, it is still much faster than forming an explicit inverse. Using `inv(A)` can take hours when the backslash operator uses minutes.

Another important thing to note is the formation of the C operator. Recall C,

$$C = \begin{bmatrix} \tilde{A}_{ii}^{-1} & -\tilde{A}_{ii}^{-1}\tilde{A}_{ie}M_{ee}^{-1} & 0 \\ 0 & M_{ee}^{-1} & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{5.19}$$

which has a large amount of zero elements. However, the blocks which are non-zero are in general very dense or unpredictable, being inverses. Since the sparse matrix format has a large overhead when creating large matrices with many nonzero elements, we instead opt to form a function which multiplies vectors with C. This both enables us to apply the earlier optimizations using the backslash operator for the lone inverses M_{ee}^{-1} and $\tilde{A}_{ii}^{-1}$, and avoids the sparse matrix overhead. The idea is to note that the matrix multiplication of such a block matrix can be written row wise:

$$Cv = \begin{bmatrix} \tilde{A}_{ii}^{-1} & -\tilde{A}_{ii}^{-1}\tilde{A}_{ie}M_{ee}^{-1} & 0 \\ 0 & M_{ee}^{-1} & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} \tilde{A}_{ii}^{-1}r_1 - \tilde{A}_{ii}^{-1}\tilde{A}_{ie}M_{ee}^{-1}r_2 \\ M_{ee}^{-1}r_2 \\ 0 \end{bmatrix} \tag{5.20}$$

Which can be then realized in MATLAB as

```
function Cr = Cxr(A_ii, M_ee, A_ie, Nf, Ni, Ne, r)
%multiply C operator with a given vector r
    Cr = zeros(Nf,1);
    Cr(1:Ni) = A_ii\(r(1:Ni) + -A_ie*(M_ee\r((Ni+1):(Ni+Ne))));
    Cr(Ni+1:Ni + Ne) = M_ee\r((Ni+1):(Ni+Ne));
end
```

which scales much better in terms of memory and speed than forming the operator explicitly. This is a general technique which should be applied to any

instances of matrices which are neither dense nor sparse, but which have blocks of both.

The rest of the operators are easy to form explicitly and while optimization may improve the runtime marginally, it is much more important to retain the clarity of the code so it closely corresponds with the treatment in section 3. The main code for the pressure solver, when operator generation is tucked away in function calls, looks like this:

```
X = restrictOperator(CG, DG, Nf);
M_ee = multiDiagonal(A_ee, A_ie);
B = formB(CG, A_ie, A_en, A_ii, M_ee);
M_nn = X*A*B;
Cr = Cxr(A_ii, M_ee, A_ie, Nf, Ni, Ne, r);

% Generate coarse rhs
q_n = X*r - X*A*Cr;
% Solve coarse system
U_n = mldivide(M_nn, q_n);

% Interpolate solution and add inn correction functions
U = (B*U_n + Cr);
% Undo the permutation to return to the original numbering
state.pressure = DG.P'*U;
if opt.Reconstruct
   % Permute system according to primal ordering
   Abar = DG.P_flux*state.A(1:Nf, 1:Nf)*DG.P_flux';
   D = formD(Abar, CG, Nf);
   % Reconstruct fine pressure
   state.pressure_reconstructed = DG.P_flux'*mldivide(D,...
            DG.P_flux*state.rhs(1:Nf) - (Abar - D)*DG.P_flux*state.
                pressure);
end
```

which is clean, readable and easy to debug[3].

5.6 Implementing iterative variants

5.6.1 Arnoldi iterations using smoothers

To implement the iterative variants, it was important that it was easy to experiment with different variants without changing large parts of the code. The steps required is the iterations themselves, which correspond to solving and

[3]Some calls related to debugging and output have been omitted for clarity

interpolating a new coarse system using the multiscale method, as well as the optional smoothing steps which remove errors near coarse edges.

Since the Dirichlet smoothers come naturally from the primal coarse grid ordering, we will use this ordering when smoothing.

```
% Ordinary MsFV iterations
fprintf('Doing MsFV %d iterations with %d sub-smoothing iterations (%s)\n', opt.Iterations, opt.Subiterations, opt.Smoother);
% Create permuted systems for the Dirichlet smoothers
Abar = DG.P_flux*state.A(1:Nf, 1:Nf)*DG.P_flux';
[D Up] = DU(Abar, CG, Nf);
rbar = DG.P_flux*(DG.P'*r);
switch lower(opt.Smoother)
    case 'dms'
        smoother = @(res) (D+Up)\res;
    case 'das'
        smoother = @(res) D\res;
    case 'jacobi'
        smoother = @(res) Abar*res;
end
error = @(Ubar)  norm(invG(r - A*(DG.P*(DG.P_flux'*Ubar))))/norm(invG(rbar));
for v = 1:opt.Iterations
    % Calculate the residual
    res = r - A*U;
    % Permute to the coarse block ordering via the original
    % ordering
    Ubar = DG.P_flux*(DG.P'*U);
    fprintf('Iteration %d:\n', v);
    for sub = 1:opt.Subiterations
        e = error(Ubar);
        res = rbar - Abar*Ubar;
        Ubar = Ubar + omega*(smoother(res));
    end
    if e<opt.Tolerance
        % Converged
        break
    end
    % Permute back
    U = DG.P*(DG.P_flux'*Ubar)
    % Perform MsFV iteration with updated pressure
    U = U + omega*invG(res);
end
```

Note the use of the relative preconditioned error for convergence testing: This is done to achieve parity with the MATLAB GMRES function which outputs preconditioned residuals. Some calls related to output and debugging have been omitted for clarity.

5.6.2 GMRES iterations

GMRES is a very useful algorithm, which guarantees convergence and numerical stability. This comes at a cost, however, as the method is significantly more difficult to implement than for example Jacobi iterations. Fortunately, MATLAB contains a standard GMRES implementation which handles general preconditioned Arnoldi systems. Since we in Section 3.7.1 noted that the MsFV iterations can be formulated as a preconditioned Arnoldi iteration, using the default MATLAB solver is no problem:

```
1 invG = @(U) Ginv(U, Ctimes, M_nn, Nf, R, X, A, B);
2 U = gmres(@(u) A*u,r,opt.Restart,opt.Tolerance,opt.Iterations,invG,
      [], U);
```

Note that we are passing a function handle for the preconditioner, instead of a preconditioner matrix, so that the more complex MsFV logic can be used instead of matrix multiplication.

Chapter 6

Theoretical performance

6.1 Performance analysis of the original formulation

We will in this section denote the fine grid size as N, the size of a coarse grid cell as $\bar{N}_c$ (approximates both dual and primal as they only differ by 1 in size) and $t(n)$ as the time spent to solve a system of n unknowns in a sparse system. We will assume that solving the linear systems dominate the running time over simple flux calculations and matrix assembly.

We will only concern ourselves with the pressure solver for this part. While reconstructing conservative flow is analogous to the constructing the pressure basis functions, we will concentrate on testing the pressure solver as it stands on its own. The analysis will be applied to the original formulation from Section 2 applied to a 2D grid before some of the properties of the operator formulation will be discussed.

Solving the full system

The cost of solving the full linear system - the time to beat - will of course be

$$t_{full} = t(N^2), \tag{6.1}$$

for a $N \times N$ fine grid. t can for example be a $O(n^1.2)$ solver.

Constructing the basis functions

We will, for each coarse cell, solve 4 basis problems (one for each corner) and one correction function. These each consist of solving a $t(\bar{N}_c^2)$ system. In addition, each of these basis functions require four boundary conditions defined by a $t(\bar{N}_c)$. We will have $\left(\frac{N}{\bar{N}_c}\right)^2$ coarse blocks to handle. This becomes

$$t_{basis} = 5\left(\frac{N}{\bar{N}_c}\right)^2 \left[t(\bar{N}_c^2) + 4t(\bar{N}_c)\right]. \tag{6.2}$$

Solving the coarse system

This is simply solving a linear equation set for the center pressure in all the inner coarse blocks.

$$t_{coarse} = t((N-2)^2/(\bar{N}_c - 2)^2). \tag{6.3}$$

Total computational cost

$$\begin{aligned} t_{msfvm} &= t_{basis} + t_{coarse} && (6.4)\\ &= 5\left(\frac{N}{\bar{N}_c}\right)^2 \left[t(\bar{N}_c^2) + 4t(\bar{N}_c)\right] + t((N-2)^2/(\bar{N}_c - 2)^2) && (6.5) \end{aligned}$$

The reconstruction step for the flow is not included in these calculations, but including it is trivial as it is simply another step of calculating another $\left(\frac{N}{\bar{N}_c}\right)^2$ sub problems.

If we are to employ the iterative MsFV steps, we must add an additional

$$t_{adaptive} = n_i\left(\frac{N}{\bar{N}_c}\right)^2 t(\bar{N}_c^2) + n_i n_{smooth} t_{smooth} \tag{6.6}$$

Where n_i is the amount of iterations, n_{smooth} the amount of smoothing steps in each iteration and t_{smooth} the cost of each smoothing iteration. t_{smooth} will

typically be low, as we will choose a cheap smoother like Jacobi iterations, which will be dominated by the matrix products for a running time of $O(N^2)$

For further analysis of the runtime of various multiscale solvers, including the MsFV-method, we refer to [KAL08].

6.2 Parallel potential

Historically, computer performance for serially executed programs has been increasing steadily over the years. For example, the Intel Pentium 2 was introduced in 1993 and had a clock speed of 60 MHz. Nine years later, in 2002, Intel's top model Pentium 3 Northwood had a clock speed of 3 GHz. Today, another nine years later, the Intel i7 CPU has a maximum speed of 3.2 GHz. While there are many subtleties in how this clock speed relates to actual execution speed, as it is highly dependent on architectural features for the family of processors, this shows how there are firm theoretical limits on the CPU speed which we are rapidly approaching. This stems from the fact that there are physical limits to how small circuits can be, since molecules are of a finite size, as well as the difficulties in dissipating the heat from so small systems.

To avoid this problem, the CPU production has shifted focus from producing CPUs with one fast core to producing several equal speed cores which execute programs simultaneously. At the same time, graphical processing units (GPUs) originally meant for computer games and 3D graphics have been re-purposed for scientific calculations. GPUs are extremely parallel devices, capable of running thousands of simultaneous threads [1]. While these approaches do not run into the problems imposed by the laws of physics, it forces us to solve computational problems differently. Giving a serial program to ten cores will not mean that the problem will be solved ten times as quickly!

There are several criterion required for a problem to be suited for parallel computing:

1. *The problem must consist of independent subproblems*: These subproblems cannot depend on each other. For example, a recursive sum where each term depends on the previous is hard to parallelize because each processing unit will have to wait until some other unit completes before it starts.

[1] Examples of such frameworks are OpenCL and CUDA. There are currently some limitations, however, such as reduced floating point precision and no integer data types.

2. *The results should reside in separate places for all subproblems*: If the processing units have to form a serial queue to store the results, and this step consists of a significant operation, performance will be lost.

3. *The subproblems must be of approximately equal size, or have a predictable performance*: If the subproblems are of different size, some processing units may take much longer to complete than others. The performance will be determined by the last processing unit to finish. If the subproblems are of different sizes, but the sizes are predictable, a good scheduler can mitigate some of the slowdown by assigning problems dynamically.

There are two main steps of the MsFV-method which are computationally intensive and are well suited for parallel programming: The construction of basis functions for the coarse pressure, correction functions and for flux reconstruction. These problems can be seen as one type of problem, since they both consist of solving subproblems on the coarse grids.

1. The generation of basis functions are fully independent of each other, with each subproblem consisting of a local part of both the input data (permeability) and the problem (solving a linear equation set).

2. The results are stored independently of each other.

3. The subproblems scale with the coarse grid, so as long as the coarse grid contains an approximately equal geometric subdivision of the global problem, each local problem will take about the same time to execute.

The step of assembling the pressure from the coarse solution and the basis functions is also well suited for parallelism:

1. The process can be done for each coarse cell, leading to independent multiplications and memory retrieval operations for each basis function.

2. The results are stored independently of each other since the coarse cells partition the domain.

3. The problem of interpolating using the basis functions scale in the same way as the generation of the basis functions.

6.3 Operator formulation

While the original formulation consists of many subproblems, the Operator formulation does away with many of the original steps and replaces them with mostly equivalent linear algebra operations. Does the parallel potential remain? We will assume, based on the results from large benchmarks on our reference implementation, that the construction of the various basis functions dominate the runtime. There are two sets of basis functions which must be generated, as well as a set of boundary problems for the pressure basis functions.

6.3.1 Pressure basis functions

In the operator formulation the pressure basis functions are constructed when A_{ii} is inverted. For this operation to be suitable for parallell/distributed computing, it is important that this operation can be broken down into smaller subproblems.

Proposition 1. *If the system matrix A of a linear system $A\mathbf{x} = \mathbf{b}$ can be permuted to a block diagonal matrix, the system can be inverted for each block separately.*

We already know from Chapter 3 that the solution of an permuted system can easily be permuted back to the original problem. However, we must prove that the block diagonal parts can be inverted separately. This is trivial; Consider the multiplication with a block inverted solution,

$$\begin{bmatrix} A_1 & & \\ & A_2 & \\ & & A_3 \end{bmatrix} \cdot \begin{bmatrix} A_1^{-1} & & \\ & A_2^{-1} & \\ & & A_3^{-1} \end{bmatrix} = \tag{6.7}$$

$$\begin{bmatrix} A_1 A_1^{-1} & & \\ & A_2 A_2^{-1} & \\ & & A_3 A_3^{-1} \end{bmatrix} = \begin{bmatrix} I & & \\ & I & \\ & & I \end{bmatrix} = I. \tag{6.8}$$

To test the structure of the generated matrices, we need to generate this permutation matrix: An simple algorithm was constructed to achieve this. The algorithm is simple and inefficient for large datasets, but it produces the correct results. Our MsFV-implementation does not employ this structure directly, since the inversion is done by built in `MATLAB` functions. For a high performance solution, many different algorithms exist for finding subgraphs in a graph which could be adapted. Decoupled systems are easy to solve for most

linear solvers regardless of this knowledge - it is only needed if we desire a distributed solution.

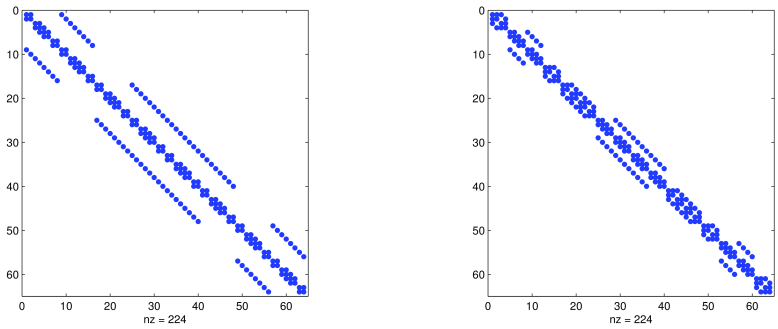
The algorithm is fairly self explanatory, and works by taking A^n where n is the size of the linear system and then finding nonzero elements in each row while doing some bookkeeping over already categorized nodes.

```
function [category P] = findblocks(A)
[N ~] = size(A);
tmp = A;
for i = 1:N
   % Use when sign calculating A^... to avoid numerical overflow
   tmp = sign(tmp*tmp);
end
% Category counter, size of picked array
cat = 1; M = 0; picked = [];
% Category of node i in position i
category = zeros(N,1);
for i = 1:N
    % Find nodes reachable from this position
    nodes = find(tmp(i,:) ~= 0);
    picked = unique([picked nodes]);
    % If the selected nodes have been updated, we have
    % a new category
    if numel(picked) > M
        %we have found a new category
        category(nodes) = cat;
        cat = cat + 1;
        M = numel(picked);
    end
    if sum(category>0) == N
        % All nodes have been categorized
        break;
    end
end
% Create new permutation matrix
c = unique(category);
ordering = [];
for i = 1:numel(c)
    ordering = [ordering find(category == c(i))'];
end
P = sparse(1:N, ordering, 1) > 0;
end
```

We will use the 2D system shown in Figure 6.1 as an example. The A_{ii} matrix of this system is shown in Figure 6.1a. While this system is, upon visual inspection, already block diagonal, there are only three obvious blocks, while our system should theoretically have nine blocks - one for each closed subset

of inner nodes. We apply the permutation matrix to the system in the regular way, $PA_{ii}P^T$, we get a clearly block diagonal structure (Figure 6.1b) which shows the geometrical properties of the system: There are nine blocks in total which come in the same order as the closed subsets of Figure 6.1. There is four smaller blocks with connections between four cells, four medium blocks with eight cells and a single large block with sixteen cells.

This property is also present for 3D systems. Because the block diagonal structure comes from each dual block's cells nodes being disconnected from every other dual block's inner cells, the system is just as easy to solve in parallel in 3D.



(a) A_{ii} (b) $PA_{ii}P^T$

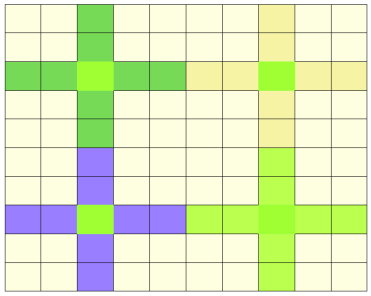


Figure 6.1: The 2D example dual grid

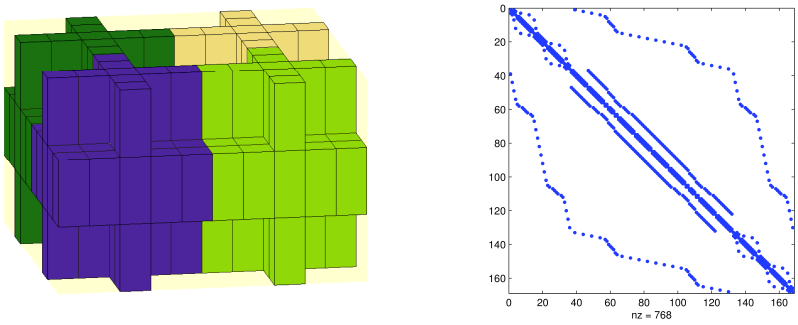
6.3.2 Boundary problems

When interpolating the solution in the operator formulation, the first step involves inverting for M_{ee}. Recall (3.23),

$$\tilde{\mathbf{u}} = \underbrace{BM_{nn}^{-1}\mathbf{q}_n}_{\text{Interpolate coarse solution}} + \underbrace{C\mathbf{q}}_{\text{Correction functions}}, \tag{6.9}$$

which describes the interpolation from a coarse pressure solution where M_{ee} also must be inverted.

The structure of M_{ee} for our 2D example system is shown in Figure 6.3a. When permuted by each subgraph in the system matrix, we get a block diagonal structure shown in Figure 6.3b. There are twelve different blocks, corresponding to each of the twelve different edges in the system. The sizes of the edges correspond to the sizes of the blocks, just as in the example for the inner nodes.



(a) A 3D example

(b) M_{ee} for the 3D example

Figure 6.2: The problem is different in 3D

When extended to the 3D example, however, the situation is different. For the dual grid shown in Figure 6.2a, the edge problem M_{ee} has a connected structure as shown in Figure 6.2b. Unfortunately, in this system all cells are connected to each other because the planes of the dual partition intersect each other. In 2D this does not happen because the central nodes in $\mathcal{I}_n$ disconnect the edges.

This may or may not be a problem for high performance implementations of the MsFV-method. While the edges should, for a big system, be a fairly minor part of the domain because the number of edge nodes grows as $O(N^2)$ when

the inner nodes grow by $O(N^3)$, this could be problematic for a highly parallel implementation. When creating a parallel algorithm, it is important that the serial parts are minimized: By Amdahl's law any serial component can have a significant impact on the speed up when many processing units are employed.

To solve this problem, a possible solution is to extend the definition of the domain decomposition 3.2 to

$$\mathcal{I}_f = \mathcal{I}_n \cup \mathcal{I}_e \cup \mathcal{I}_i \cup \mathcal{I}_s. \tag{6.10}$$

Where $\mathcal{I}_s$ is the subset where the different edges intersect each other. When this is done, redoing the steps in Chapter 3 should lead to a system where instead of interpolating along the edges from the central nodes, the interpolation will instead go from the central nodes to the nodes $\in \mathcal{I}_s$ and further to the rest of the edges, leading to the inversion of two block diagonal systems instead of a single none-block diagonal system.

If we permute the system analogously to the earlier treatment with the new indices included, the only differences in the method will be the B and C operators. We will disconnect A_{ss} from the neighbouring edge nodes in the same manner as with A_{ee},

$$M_{ss} = \tilde{A}_{ss} + \text{diag}\left[\sum_i \tilde{A}_{se}^T\right]. \tag{6.11}$$

Once this is done, we can again block eliminate a reduced linear system,

$$M = \begin{bmatrix} A_{ii} & A_{ie} & 0 & 0 \\ 0 & M_{ee} & A_{es} & 0 \\ 0 & 0 & M_{ss} & A_{ns} \\ 0 & 0 & 0 & A_{nn} \end{bmatrix}, \tag{6.12}$$

leading to the new operators

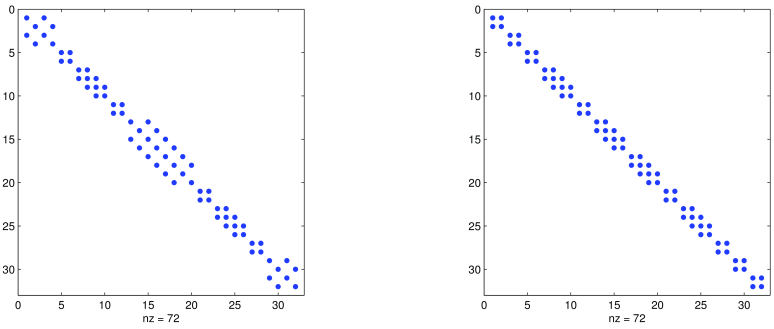
$$B = \begin{bmatrix} A_{ii}^{-1} A_{ie} M_{ee}^{-1} A_{es} A_{sn} M_{ss}^{-1} \\ M_{ee}^{-1} A_{es} A_{sn} M_{ss}^{-1} \\ A_{sn} M_{ss}^{-1} \\ I \end{bmatrix} \tag{6.13}$$

and

$$C = \begin{bmatrix} A_{ii}^{-1} & -A_{ii}^{-1} A_{ie} M_{ee}^{-1} & A_{ii}^{-1} A_{ie} M_{ee}^{-1} A_{es} M_{ss}^{-1} & 0 \\ 0 & M_{ee}^{-1} & -M_{ee}^{-1} A_{es} M_{ss}^{-1} & 0 \\ 0 & 0 & M_{ss}^{-1} & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}. \tag{6.14}$$

While more complex in terms of representation, the structure of the matrices is straightforward: The general pattern of solving a problem and then extrapolating to a new set of nodes before repeating occurs many times in the solved system. The non-zero parts of the original C matrix can be found in the upper left parts of the new C matrix.

This altered method has the advantage of having much smaller condition numbers as well as approximately constant condition numbers for constant coarse sizes. More importantly, it makes the edges possible to be inverted fully independently of each other just as the inner nodes! While for some cases the error is somewhat increased because of the additional localization assumptions, the increased speed for large data sets, as well as the fantastic parallel potential makes up for it as we will see in Section 6.3.4. For 2D, the two methods coincide, as no edge boundaries overlap.



(a) M_{ee} (b) $PM_{ee}P^T$

Figure 6.3: The permuted 2D edge matrix M_{ee}

6.3.3 Conservative flow basis functions

When looking at the step required for constructing a conservative pressure field, it becomes apparent that the D matrix which must be inverted already exists in a block diagonal form (Figure 6.4). This comes from the fact that each primal coarse block is disconnected when creating D, which is also lexicographic in the indices of $\bar{\Omega}_i$.

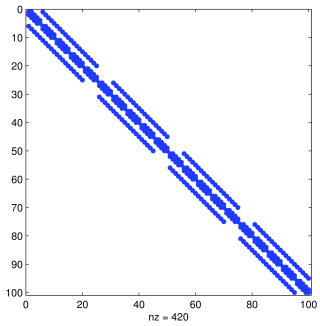


Figure 6.4: The matrix D for the flow basis functions is already block diagonal

6.3.4 Condition numbers for M_{ee}, A_{ii} and D

The condition number of a matrix is a measure of how small changes in the right hand side of the system $A\mathbf{x} = \mathbf{b}$ change the values of $\mathbf{x}$. A smaller condition number will result in smaller changes from these perturbations and for many numerical linear algebra algorithms the condition number has a significant influence on the speed and accuracy of the solution. Generally, we hope for matrices with a low condition number. A block diagonal matrix, which we have shown to be solvable blockwise, should have a low condition number, since the changes in the right hand side need not influence all the block solutions, again reflecting that linear solvers can take advantage of the block structure without being strictly aware of it.

For a normal matrix we know that the condition number $\kappa(A) = |\lambda_{max}/\lambda_{min}|$ and since the matrices are symmetric, they are also normal.

For a block diagonal matrix, the eigenvalues are equal to the eigenvalues of all the blocks because the characteristic polynomial for the block matrix is

$$\left| \begin{bmatrix} A_1 - \lambda I & & \\ & A_2 - \lambda I & \\ & & A_3 - \lambda I \end{bmatrix} \right| = |A_1 - \lambda I||A_2 - \lambda I||A_3 - \lambda I|, \qquad (6.15)$$

which has zeroes at all the eigenvalues of the blocks. Because of this, for coarse partitions of constant size with uniform permeability, we should have the same condition number for A_{ii} and D regardless of the number of fine cells. This should also apply to M_{ee} in 2D, but not in 3D. When run on a series of different cases with constant coarse sizes, the experiments seem to confirm the assump-

tions (Table 6.1). The cases have a log normal permeability distribution and constant coarse sizes 10×10 and $10 \times 10 \times 10$ for 2D and 3D respectively. The condition numbers are calculated for subsets of the left hand side of the multiscale system and are as such independent of boundary conditions and well configurations.

The condition numbers for the M_{ee} system are the most interesting: With the original formulation, the numbers are much larger and will steadily increase with the problem size, but with the improved method derived in Section 6.3.2, the condition numbers are much smaller and constant.

The condition numbers for M_{ee} and A_{ii} should depend on the ratio between the smallest and largest permeability in the corresponding fine cells: If there are large variations, the condition numbers will be larger, and iterative methods will take longer to converge. To verify this, a $50 \times 50 \times 50$ fine grid with a $5 \times 5 \times 5$ coarse grid was generated. The permeability for every second cell was set to 1 and the rest to a lower value, varying from 10^{-1} to 10^{-10}. The resulting condition numbers can be seen in Figure 6.5. From the plot it is obvious that the larger variations lead to larger condition numbers. The improved method has yet again a much lower condition number for M_{ee}, here marked with an asterisk.

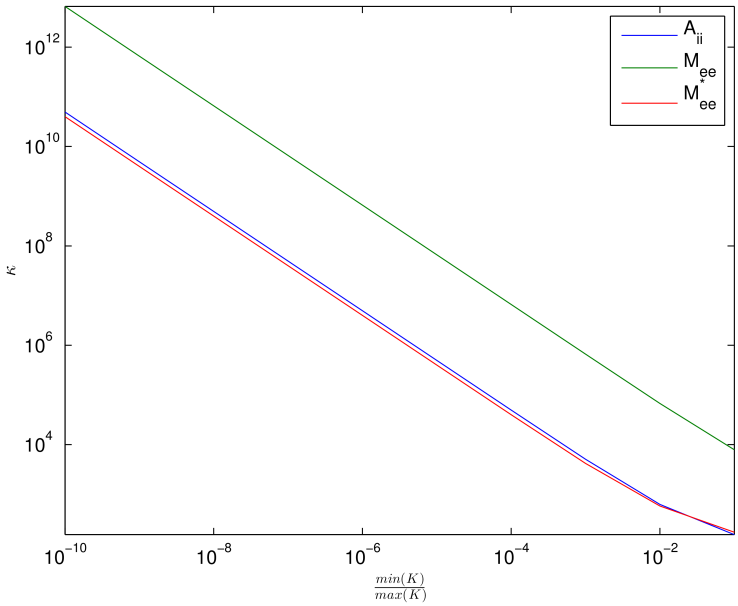


Figure 6.5: Condition numbers as a function of permeability variation. M^*_{ee} is the improved formulation

Table 6.1: For constant coarse size, the condition numbers of some sub matrices are constant. The improved method has much lower condition numbers for the edge problems.

(a) Condition numbers in 2D

System size	$\kappa(A_{ii})$	$\kappa(M_{ee})$	$\kappa(D)$
50×50	46.3	144.6	58.5
100×100	46.3	144.7	58.5
150×150	46.3	144.7	58.5
200×200	46.3	144.7	58.5
250×250	46.3	144.7	58.5

(b) Condition numbers in 3D

System size	$\kappa(A_{ii})$	Original $\kappa(M_{ee})$	Improved $\kappa(M_{ee})$	$\kappa(D)$
$30 \times 30 \times 30$	52.2	705.7	57.8	85.3
$40 \times 40 \times 40$	52.2	942.1	57.9	85.3
$50 \times 50 \times 50$	52.2	1144.2	57.9	85.3
$60 \times 60 \times 60$	52.2	1264.0	57.9	85.3
$70 \times 70 \times 70$	52.2	1347.8	57.9	85.3

Chapter 7

Results

7.1 Intro

Implementing a new method is only part of the job: A thorough verification process is needed to map out the advantages and disadvantages of any new method. For this, we need some preliminaries which should be familiar to anyone experienced in numerical analysis.

The relative difference will be used for the error analysis. We will define the error vector $\mathbf{e}$ as

$$\mathbf{e}_i = \mathbf{P}_i - \tilde{\mathbf{P}}_i \tag{7.1}$$

where $\tilde{\mathbf{P}}$ and $\mathbf{P}$ are the multiscale and reference solutions respectively. This results in a relative error which is independent of problem scaling. We will furthermore use the relative l^2 norm,

$$\|\bar{\mathbf{e}}\|_2 = \frac{\|\mathbf{e}\|_2}{\|\mathbf{P}\|_2} = \frac{\sqrt{\sum_{i=0}^{N} \mathbf{e}_i^2}}{\sqrt{\sum_{i=0}^{N} \mathbf{P}_i^2}} \tag{7.2}$$

as well as the relative maximum norm,

$$\|\bar{\mathbf{e}}\|_\infty = \frac{\|\mathbf{e}\|_\infty}{\|\mathbf{P}\|_\infty} = \frac{\max\limits_{0 \le i < N} (|\mathbf{e}_i|)}{\max\limits_{0 \le i < N} (|\mathbf{P}_i|)}. \tag{7.3}$$

Because point wise error can be seen as a fairly strict measurement in itself, both streamlines and pressure plots will be produced. When comparing plots of reference and MsFV solutions, the colorbars will be synced to ensure the best possible visual comparison.

For error plots, we will use the point wise absolute error scaled by the total variation in the problem,

$$\frac{|\mathbf{e}_i|}{|max(\mathbf{P}) - min(\mathbf{P})|}. \tag{7.4}$$

7.2 Permeability generation and fluid type

For our synthetic examples, we will need to generate permeability for our geometries. To do this, we will use the Carman-Kozeny relation,

$$K = \frac{1}{8\tau A_v^2} \frac{\Phi^3}{(1-\Phi)^2} \tag{7.5}$$

which relates porosity Φ, tortuosity τ and the specific surface area A_v to the permeability K. The tortuosity is the square root of the ratio between the length of the average flow path and the distance between the arc points. We will not delve deeper into this subject, but rather note that the approximation is meant for flow with a low Reynolds number, just like Darcy's law. For values we will use $A_v = 6/(10\mu m)$ $\tau = 0.81$.

For the porosity we will use a Gaussian random field with standard deviation 2.5, minimum value 0.2 and maximum value 0.4, unless otherwise is noted. When generating layered permeability,

7.3 2D validation

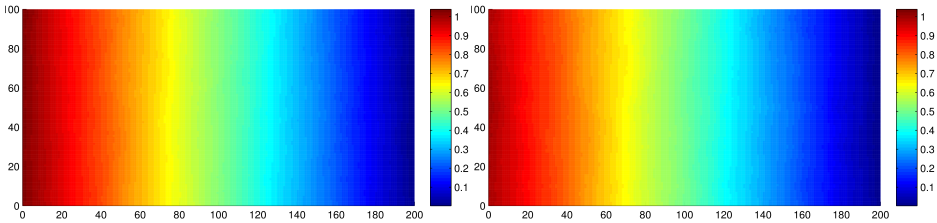
Simple 2D cases are a good candidate for early validation of the method because it makes it trivial to visualize the entire domain. While 3D is the most interesting case for real life computational examples, all of the previous work done on the MsFV-method has been validated in 2D, which makes it important to have a 2D solver for validation purposes.

There are two ways to impose pressure differentials upon a system: Wells and boundary conditions. For the basic 2D validation, we will employ a grid consisting of 200×100 fine cells and 10×5 coarse grid for a total of 400 fine cells

in each coarse block. For these examples the previously mentioned Carman-Kozeny relation will be used to generate the permeability distribution.

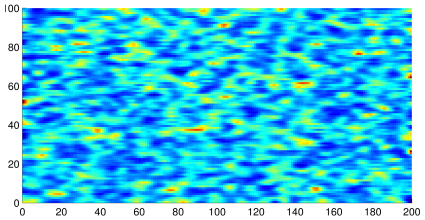
7.3.1 Flow channel

The first example is a simple flow channel, with a total of $1m^3$ units of fluid being injected at the left edge (Neumann boundary) at $x = 0$, and zero pressure boundary condition along the right edge (Dirichlet boundary) at $x = 200$, with no flow-boundary along the top and bottom faces at $y = 0$ and $y = 100$. This should give a steady flow from the left to right, with a monotonically decreasing pressure to go along with it. The domain has size $L_x = 200$ and $L_y = 100$.

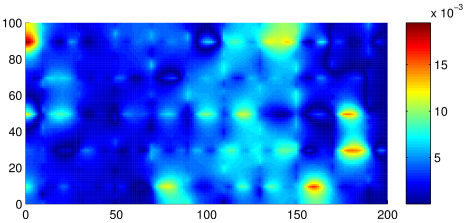


(a) Reference

(b) Multiscale solution



(c) Permeability



(d) Error

Figure 7.1: Results for the two dimensional flow channel example with both Neumann ($x = 0$) and Dirichlet ($x = 200$) boundary conditions

The results can be seen in Figure 7.1. The multiscale pressure solution 7.1b is qualitatively very close to the reference solution 7.1a and it is hard to spot any significant difference. When looking at the error plot in Figure 7.1d, however, the error is significantly larger near the edges of dual volumes. For instance, there are five primal coarse blocks along the y-axis at $y = 0, 20, 40...$ which

results in the dual grid having edges at $y = 10, 30, 50, ...$

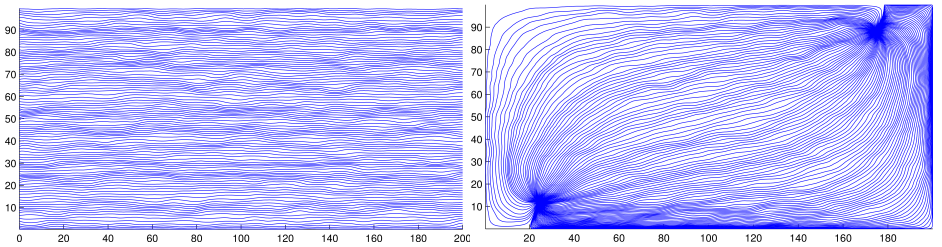
Generally, the error is always largest near the edges of the dual grid because of the localization assumption, but if there are large differences in permeability along the edges, the edge solutions will be very bad at interpolating correctly, since a solution accounting for the entire grid would simply flow around the impermeable block if there was low permeability around it, leading to reduced pressure drop over the impermeable cell. This can be illustrated as in Figure 7.2a, where a single impermeable block leads to very little flow and a large pressure drop, but the same system when the neighbouring cells are included leads to a much smaller pressure drop as in Figure 7.2b. Clearly, the localization assumption can fail in terms of the point wise error in the system - the average error is still small, $\|\bar{\mathbf{e}}\|_2 = 0.026$, or less than three per cent.

The expected flow pattern is shown in Figure 7.3a and, as expected, is a steady flow from left to right, curving around the areas with high permeability. When constructing a flow field from the non-smooth initial pressure solution, shown in Figure 7.3c we can see that the discontinuities of the derivative becomes problematic: The pressure lines end up being non-smooth and occasionally sharp 90 degree turns are produced in the flow pattern. Starting lines at $x = 0$ wind up being clustered close together and the streamlines are obviously not physically realistic. This is as expected, considering how the pressure is non-smooth at all dual cell boundaries, which makes any attempt to construct a flow field mathematically problematic - flow induced by the change in pressure across coarse blocks is not well defined.

When using the reconstructed pressure field meant for flow fields, the results are much better (Figure 7.3e): Qualitatively speaking, the patterns show the expected curving around high permeability areas, as well as the areas where the flow lines are very close to each other (for instance at $y = 90$).

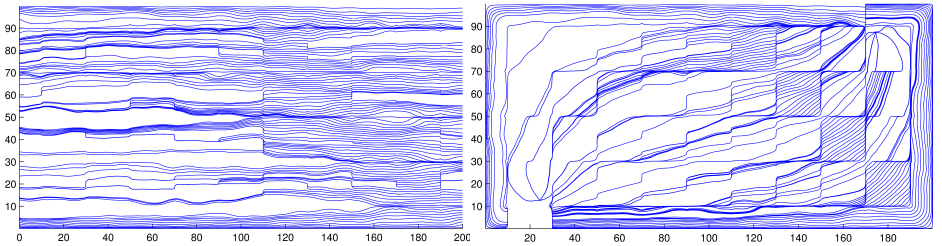
(a) Thin edge (b) Double edge

Figure 7.2: A single impermeable cell can pose problems for the localization assumption when interpolating pressure values



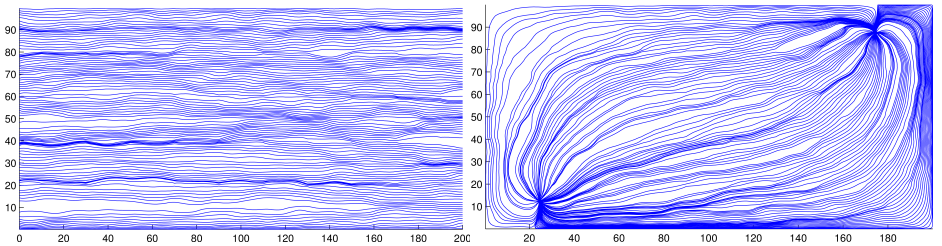
(a) Reference flow, channel

(b) Reference flow, 2 wells



(c) Multiscale flow, channel

(d) Multiscale flow, 2 wells



(e) Conservative multiscale flow, channel

(f) Conservative multiscale flow, 2 wells

Figure 7.3: Results for the 2D flowchannel example

7.3.2 Two wells

To verify that both rate and BHP wells behave correctly in 2D, we will solve a problem with one of each type placed at $(L_x\frac{1}{8}, L_y\frac{1}{8})$ and $(L_x\frac{7}{8}, L_y\frac{7}{8})$ so that the lower left corner has a rate well pumping in $1m^3/s$ units of fluid and the lower right corner has a BHP well which enforces zero pressure. Combined with no-flow conditions along the boundary this should lead to a steady change in pressure from the lower left corner to the upper right, as shown in 7.4a.

The multiscale solution 7.4b is qualitatively very similar, but seems to have higher error near the wells themselves, as shown in Figure 7.4d. This is not unexpected, as the treatment of wells in the MsFV-method has some problems modelling a strict point well: The wells are first added in an integral sense to the system (distributed over a coarse block), while the correction functions are used to reduce the error of the integral approximation. The error again shows the dual coarse boundaries, illustrating that the choice of dual grid can influence the quality of the solution.

While the approach to point wells leads to some error around the wells themselves with the largest relative error being large near the BHP well, the relative error is very low at $\|\bar{\mathbf{e}}\|_2 = 0.041$. This is in general quite a good result because while the error near the wells is significant, the general smoothness of pressure solutions makes this unproblematic for the rest of the domain.

When observing the flow patterns, we again see that the reference flow (7.3b) is very similar to the flow from the reconstructed pressure field (7.3f). The multiscale flow (7.3d) shows the fact that the non-smooth pressure does not lead to meaningful flow fields - the streamlines end up alongside the edge dual volumes in many places since the discontinuities act as a barrier for the flow. Flow lines also intersect each other near the wells, which should never happen for incompressible flow.

7.4 Effects of coarse grid selection

7.4.1 Variations in permeability

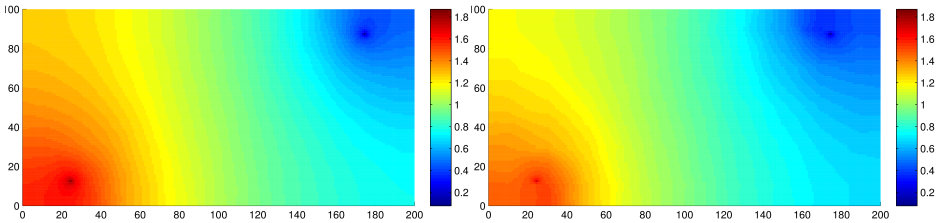
It is important to note that the selection of the coarse grid can influence the solution. We first saw this in Section 7.3.1 where a coarse dual block boundary intersected both high permeability and low permeability cells leading to a high local error. The assumptions made by the algorithm is not always valid. For example, take a domain consisting of low, uniform permeability everywhere except a small circle near the center of the domain where the permeability is 10^6 Darcy. The low permeability continues on the inside of the circle as shown in Figure 7.5b. We will reuse the boundary condition from Section 7.3.1, inducing flow across the domain. A coarse grid of 3×3 is used.

The result is interesting: The error, shown in Figure 7.5c, is low in the domain outside of the circle. This is unsurprising, as this is the same problem we

solved earlier. Inside the circle, however, the error is very large: Even though there should be very low pressure inside the impermeable circle since there are no source terms there, there is a significant pressure increase from the edge of the wall to the center.

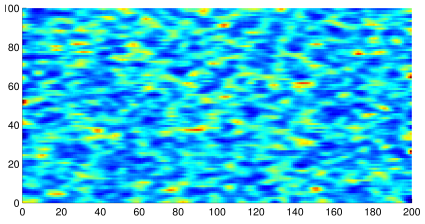
Why does this happen? The example has been constructed so that the circle is entirely within the central primal coarse block $\bar{\Omega}_i$ as shown in Figure 7.5a. This results in the center point of $\bar{\Omega}_i$ being inside circle. When constructing the linear system for finding the pressure at this center point, the MsFV-method estimates the flux across the boundaries of $\bar{\Omega}_i$ induced by the basis functions which intersect $\bar{\Omega}_i$. There is obviously no induced flow from the basis functions corresponding to unit pressure at the center node because of the circle, but the other nodes will result in flux into $\bar{\Omega}_i$. Since this flux sum estimates that there should be positive pressure at the center node, this value is then interpolated using the basis functions.

The basis functions prevent there from being pressure outside of the circle, but the unexpected result of there being pressure *inside* an impermeable cell with no source terms remains. Even though this is a synthetic example, it shows that the localization assumption can lead to unpredictable results when there

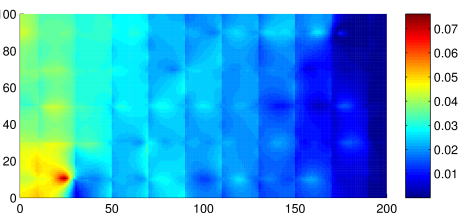


(a) Reference

(b) Multiscale solution



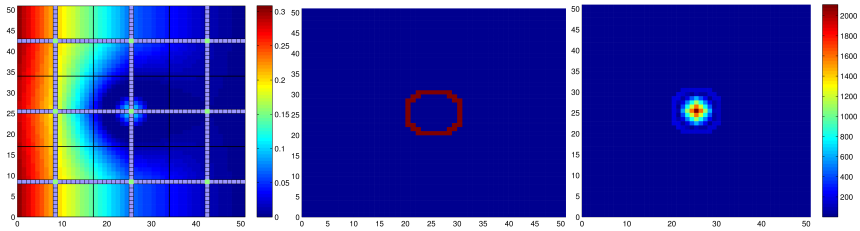
(c) Permeability



(d) Error

Figure 7.4: Results for the 2D flowchannel example

is great variation in permeability within a coarse block.



(a) Multiscale solution (b) Permeability (c) Error

Figure 7.5: Results for the 2D flowchannel example

7.4.2 The thickness of dual boundaries

The thickness of the dual boundaries is difficult to manage. For some grids is is difficult to ensure a dual grid two cells thick everywhere and for unstructured grids the partitioning schemes almost always result in two or three cells thickness in some places. However, this is not actually a problem. While the description of the original algorithm from Chapter 2 does not describe how to handle this situation, the operator formulation in Chapter 3 handles this implicitly by being a pure linear algebra formulation: The solution is extrapolated from coarse center points using the edge nodes, but there are no restrictions placed on the edge nodes in terms of thickness.

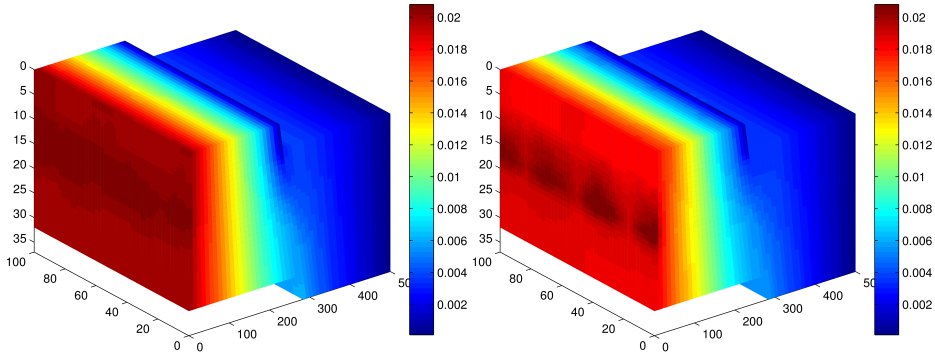
There is of course an extra computational cost associated with the solution of a bigger linear system for the edge nodes' basis functions, but numerical experiments indicate that the difference between a dual grid with two cell thickness and one cell thickness is negligible compared to the difference in error caused by how the dual grid overlaps the underlying permeability. There are even some cases where having a double thickness can be advantageous for the error, for example in the situation illustrated in Figures 7.2a and 7.2a.

7.5 3D validation

7.5.1 Flow channel with fault

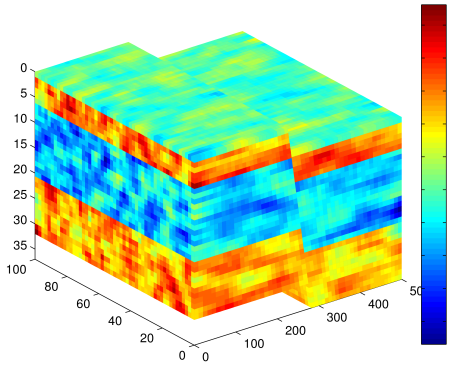
To validate the 3D implementation, a simple sloped fault will be used with lognormal permeability layers with mean values 100, 400, 50 and 350 in each layer, emulating the distribution found in sedimentary rocks. The standard deviation is 4.5. The flow channel boundary condition from the 2D validation will be reused. The dual grid for this fault was shown in Section 5.3.2.

The permeability is shown in Figure 7.6c and the reference and multiscale solutions can be seen in Figure 7.6a and 7.6b respectively. The error seems to be slightly higher near the changes in permeability and near the fault, but in general, the whole domain has an error comparable to the same example run without a fault. The relative error is fairly low, at $\|\bar{\mathbf{e}}\|_2 = 0.094$. This indicates that the handling of faults with the planar algorithm is good, something which has not been achieved before in research on the method. Having a method which is robust in regards to faults and other discontinuities in the grid is extremely important, as such features will be found in almost all realistic reservoir data sets. The fact that the algorithm automatically creates dual grids capable of handling faults without being explicitly told that the fault is there is also promising.

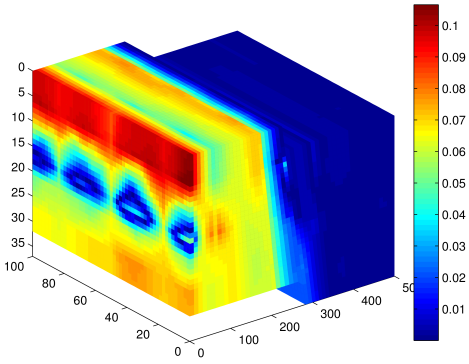


(a) Reference

(b) Multiscale solution



(c) Permeability



(d) Error

Figure 7.6: Results for the 2D flowchannel example

7.6 Realistic datasets

Since the results for Cartesian grids and synthetic permeability cases were good, we will attempt to handle both realistic geometry and realistic permeability. The first test consists of two subsets of the highly challenging SPE10 dataset, which provides a difficult permeability configuration in a Cartesian geometry and the second test will be an example of realistic geometry in the Johansen dataset.

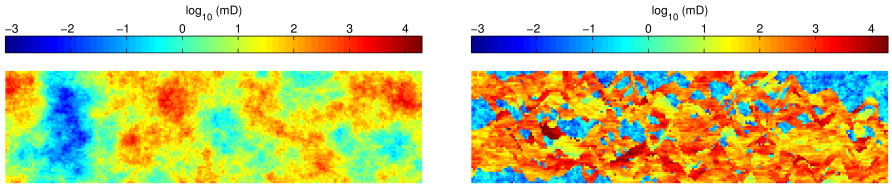
The examples chosen are not used because they represent problems the MsFV-method solve without any problems; rather, they have been selected so that the solutions will illustrate the weaknesses of the method.

7.6.1 The SPE10 dataset

For an example of a highly heterogeneous permeability, we will turn to the SPE10 dataset. The SPE10 dataset was originally published by the Society for Petroleum Engineers in 2001[CB01] with the purpose of comparing different upscaling techniques on a very challenging dataset. Since multiscale methods use solutions on several scales to get results, they are closely related to traditional upscaling methods, and the SPE10 data set is a good candidate pushing the MsFV-method to its limits.

The SPE10 dataset consists of $60 \times 220 \times 85$ fine cells and can be divided into two parts: The upper 35 layers are called the *Tarbert formation* and consists of a highly challenging layered permeability. The lower 50 layers are called the *upper Ness layers* and contains several channel-like structures in the permeability. An example layer from Tarbert and Ness can be found in Figures 7.7a and 7.7b respectively. Hollow visualizations of the permeability in horizontal (Fig. 7.8b) and vertical (Fig. 7.8c) directions, as well as the porosity (Fig. 7.8a) are provided in Figure 7.8.

The physical dimensions in the xy-plane are 1200×2200 feet or about 365×670 meters. We will use the well set up from the original testing environment, shown in Figure 7.7a. There are four producer wells, which are kept at 200 bars pressure near the corners and a single injector well with 500 bars pressure. All wells are 12.5 cm in radius and penetrate through all the layers. For both cases, we will consider a subset of 30 layers, with sampling for plots being done in layer 22 and 12 to avoid the coarse boundaries. The coarse grid will consist of $5 \times 5 \times 5\times$ fine cells, in accordance with the SPE10 example from



(a) Tarbert (b) Upper Ness

[LTL10]. Informal tests suggest that much bigger coarse blocks for a case with this degree of anisotropy leads to completely unusable solutions.

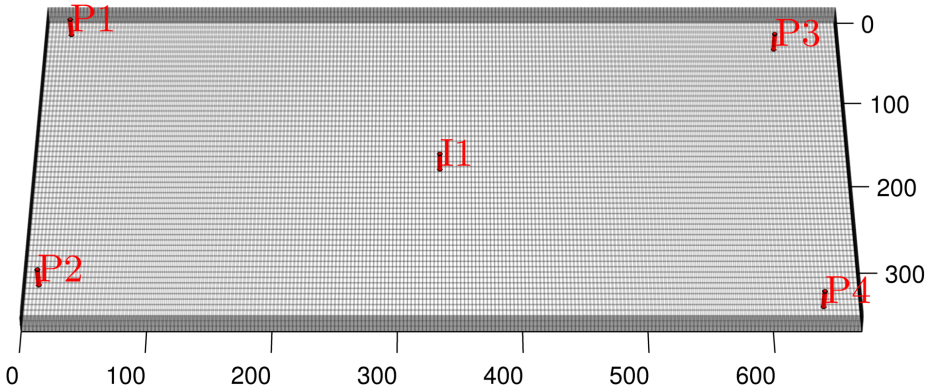
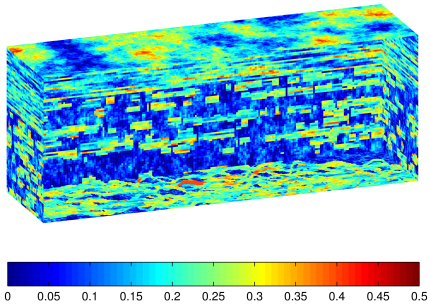
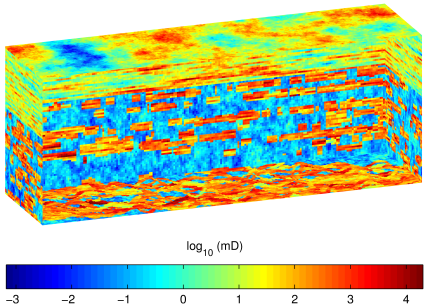


Figure 7.7: The four producer wells (P1-4) and the injector well (I1) in the SPE10 subset.

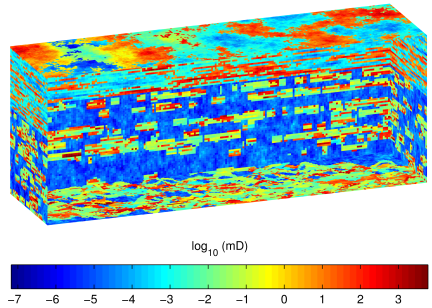
The aspect ratio combined with the highly heterogeneous environments is extremely challenging and non-iterative variants of the MsFV-method typically struggles with such setups because of the highly non-local features of the pressure, with both channels and greatly varying permeability [KAL08]. The fact that the placement of the wells can have a large influence on the pressure distribution because of the large variance in permeability does not bode well, as the well handling in the MsFV-method partially considers wells in an integral sense when constructing the right hand side of the coarse system.



(a) Porosity



(b) Horizontal permeability



(c) Vertical permeability

Figure 7.8: SPE10 geological data

Table 7.1: Error and runtime for the Tarbert layers

	Normal	Improved
$\|\bar{\mathbf{e}}\|_2$	2.039	0.040
$\|\bar{\mathbf{e}}\|_\infty$	276.321	0.893
$t_{operators}$	8m32s	1m27s

Tarbert formation

The Tarbert formation, as exemplified by Figure 7.7a, has a highly heterogeneous lognormal permeability distribution with a range spanning 10^7 mD. The results can be seen in Figure 7.10, where the original operator formulation (7.10b and 7.10d), the decoupled formulation (7.10a and 7.10c) can be seen together with the reference solution (Figures 7.10e and 7.10f).

The results are overall good, as is to be expected considering that for small coarse volumes, the local permeability will be fairly uniform. The pressure drop around the BHP wells in the corners seem underestimated in the MsFV solutions, most likely because of the difference in well handling combined with the differences in local permeability. The lower right corner of both MsFV solutions at layer 12 end up with a large pressure increase near the well where the solver fails. This is much more pronounced in the solution without speedup. For layer 22, the solution quality is excellent, except in a small neighbourhood around the corner wells.

In terms of the error, there are some surprises. The results in Section 7.6.2 seem to imply that the speedup method gains greater speed at the cost of accuracy, but the results for the benchmark seem to indicate that the new method is significantly better for the lower layers of SPE10, as shown in Table 7.1, as well as significantly faster by a factor of about 5.

A possible explanation for the disparity is that the shale layers complicate the solution process, but this does not explain why the error is slightly larger for the new method when applied to simple 3D examples such as case 1. The highly non-uniform permeability of SPE10 could make the edge problems easier to solve independently of each other instead of as a single coupled system because of high variations in permeability intersecting the edges.

The streamlines for the Tarbert case is shown in Figure 7.9a. A streamline originates in the center of all the cells for the central producer well to ensure that the flow pattern is captured. The original solution has red streamlines and

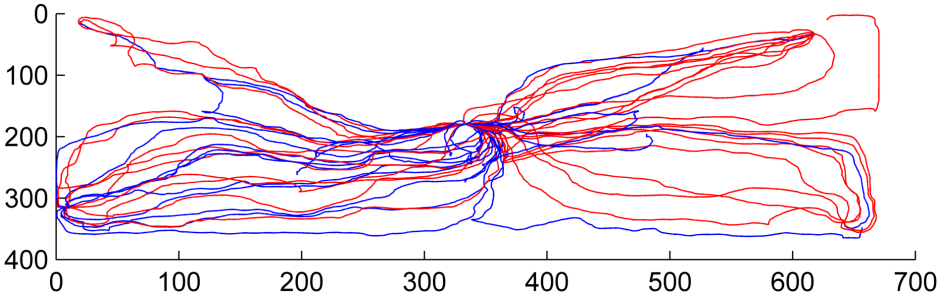
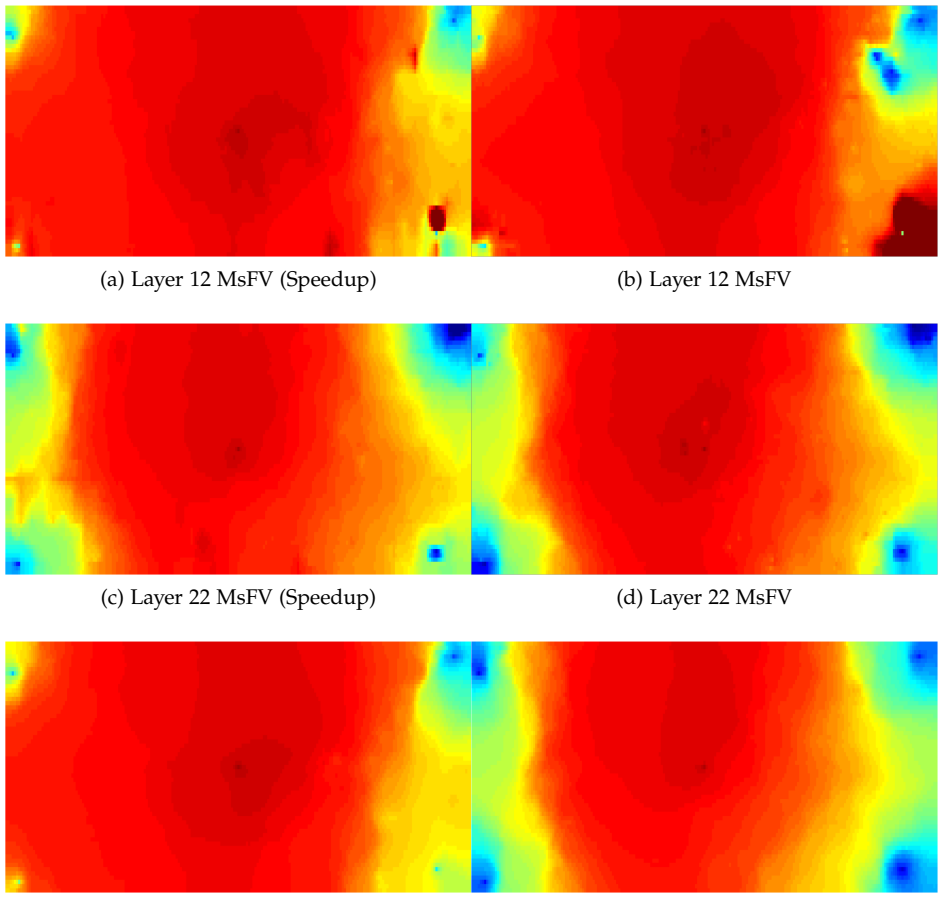


Figure 7.9: Streamlines for the Tarbert subset. Red comes from the TPFA reference solution and blue from the MsFV approximation, after pressure reconstruction

the MsFV solution is shown in blue. On the positive side, the flow pattern seems very similar in terms of where the flow goes. While there seems to be fewer streamlines to the wells on the right hand side, this could be explained by the great local variations in permeability, which makes the streamlines sensitive to small perturbations and the slightly different well handling of the multiscale method.

All in all, the MsFV handles the Tarbert formation fairly well. For the most part, the solutions are visually indistinguishable and the flow patterns seem somewhat similar. The fact that the faster method is also the best at solving this formation is surprising and very interesting.



(a) Layer 12 MsFV (Speedup)

(b) Layer 12 MsFV

(c) Layer 22 MsFV (Speedup)

(d) Layer 22 MsFV

(e) Layer 12 TPFA Reference

(f) Layer 22 TPFA Reference

Figure 7.10: The results for the Tarbert SPE10 layers

Table 7.2: Error and runtime for the upper Ness layers

	Normal	Improved
$\|\bar{\mathbf{e}}\|_2$	16.631	2.457
$\|\bar{\mathbf{e}}\|_\infty$	2355.499	120.324
$t_{operators}$	8m41s	1m30s

Upper Ness

The Ness layers are the most challenging part of the SPE10. As can be deducted from Figure 7.7b there are two different distributions of very low (shown in blue) and very high permeability (shown in red). The aspect ratio of each of these channels is also large, as some channels run for the entire length of the model with a thin thickness. Highly anisotropic problems is known to be a problem for the MsFV-method [KAL08] and this seems to extend to 3D: The original formulation is shown in Figures 7.12b and 7.12d and the improved formulation is shown in Figures 7.12a and 7.12c, and the results are poor. It is not easy to tell which version of the method fares better from the plots, but Table 7.2 reveals that the improved method is significantly better both in runtime and error wise, but still too high to be counted as a successful solve.

The streamlines, in Figure 7.11a, flow towards the wells, but most terminate before reaching a well because of the poor solution quality.

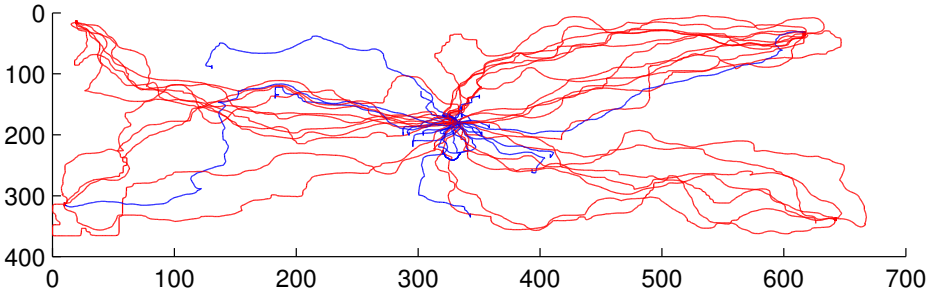
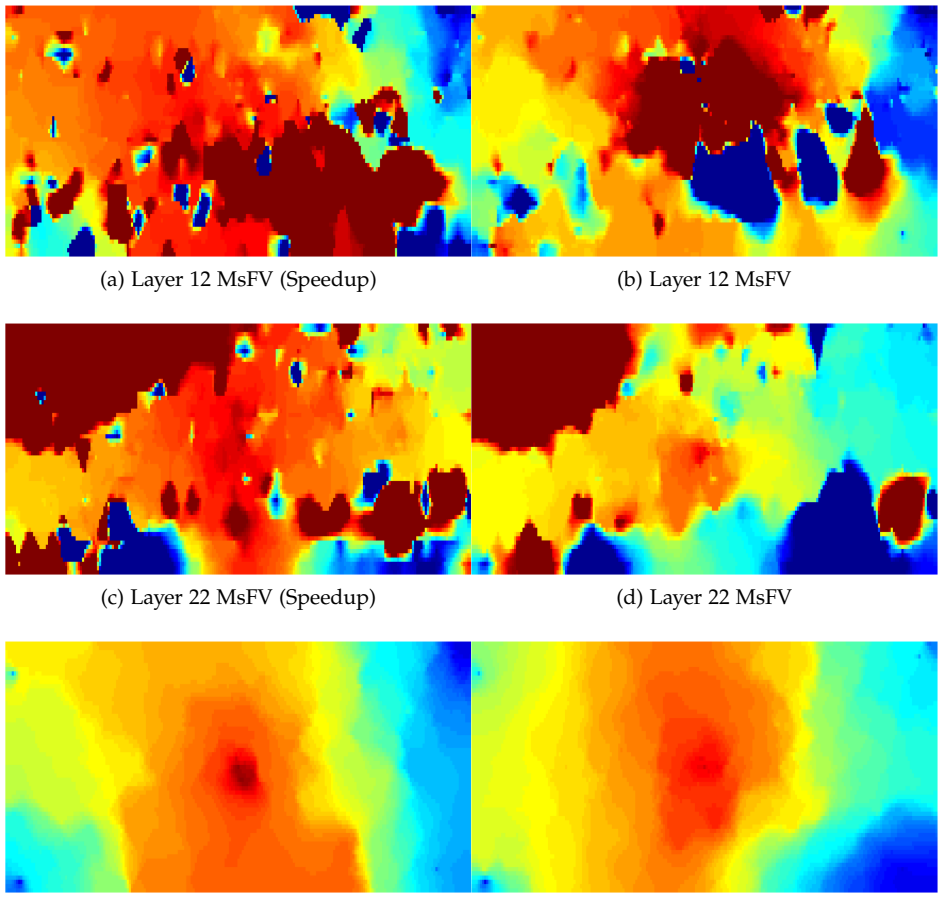


Figure 7.11: Streamlines for the Ness subset. Red comes from the TPFA reference solution and blue from the MsFV approximation, after pressure reconstruction



(a) Layer 12 MsFV (Speedup) (b) Layer 12 MsFV

(c) Layer 22 MsFV (Speedup) (d) Layer 22 MsFV

(e) Layer 12 TPFA Reference (f) Layer 22 TPFA Reference

Figure 7.12: The results for the upper Ness SPE10 layers

7.6.2 The Johansen formation

The Johansen dataset is a corner point grid constructed for the purpose of evaluating possible large scale CO_2 storage underneath the ocean. It is publicly available [EDH+09] and an example of a corner point grid with realistic permeability. The model consists of a subset of the Johansen formation, which is embedded between two shale layers with very low porosity and permeability. In the current form, the model is based on both 2D and 3D seismic data, as well as data extrapolated from similar reservoirs. Although the dataset is meant for CO_2 storage simulation, it is also a publicly available model suitable for regular reservoir simulation. While not representing any actual well system this enables us to create easily reproducible results on realistic data sets.

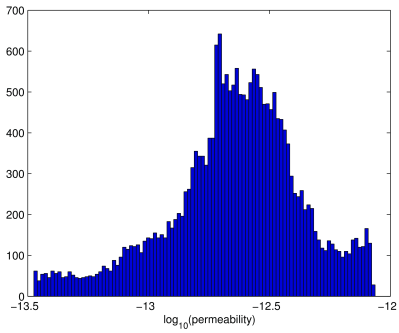
The subset we will use is shown in Figure 7.13a For our pressure solver, we are interested in the actual rocks inside the shale layers, which can be visualized by filtering away the very low porosity as shown in Figure 7.13b. Note that while we will use the porosity to filter away parts of the data set, the porosity is not in use when solving for incompressible flow. It is in this case, however, correlated with permeability and is therefore convenient to use to remove the shale layers with very low permeability. As can be seen from the plots, the geometry contains complexities typical of realistic data sets: A big fault as well as bumps and non-smooth surfaces. The permeability for the actual rocks is distributed as shown in Figure 7.13a and has a distribution which looks normally distributed in a range of about 30 in order of magnitude.

We will employ a $15 \times 15 \times 3$ dual grid for the simulation. This grid, with the horizontal part of the dual grid stripped away for clarity, can be seen in Figure 7.13b. The planar partitioning algorithm handles the corner point grid as expected from the earlier experiments.

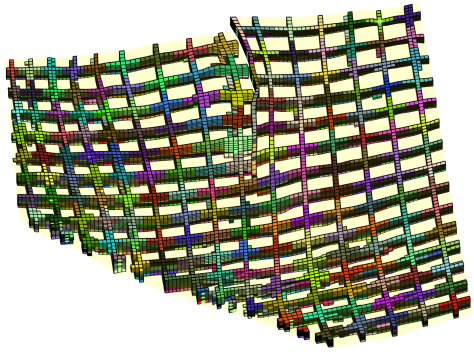
The results for the shale layers are not too good - for both the original MsFV-method (Figure 7.16a) and the new formulation (Figure 7.16b) the error is significant. Some areas overestimate the pressure drastically and others underestimate the pressure to zero. This probably happens because of the extremely low porosity, which makes it difficult to create good quality pressure basis functions since there is very little flow going on.

When looking at the inner layer with more normal porosity levels, however, the results are much better. The original formulation (Figure 7.15b) successfully captures the solution from a qualitative perspective[1]. This is shown in the

[1]Because of this, the reference solution has been omitted to save space



(a) The permeability distribution for the rocks in the Johansen dataset, given in Darcy

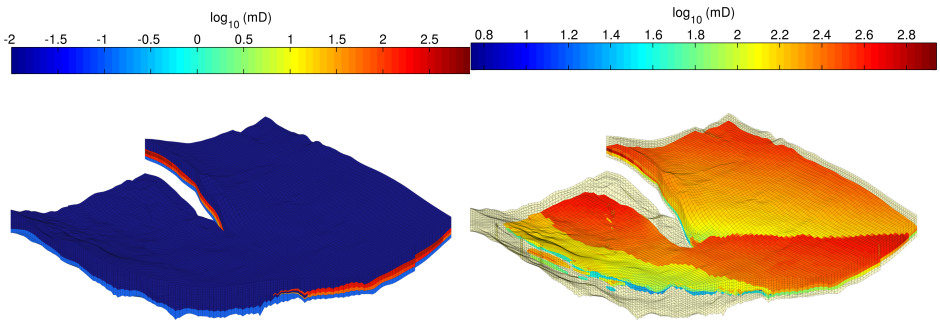


(b) Dual grid without the horizontal parts

Table 7.3: Johansen error

	Normal (inner)	Improved (inner)	Normal (outer)	Improved (outer)
$\|\bar{\mathbf{e}}\|_2$	0.030	0.354	1.660	13.2047
$\|\bar{\mathbf{e}}\|_\infty$	0.234	16.023	195.174	1285.423
$t_{operators}$	11.5s	6.6s		

error values shown in Table 7.3 as well as in the error plots (Figure 7.15d & 7.15c): The error is low for the normal MsFV solution ($\|\bar{\mathbf{e}}\|_2 = 0.03$) and significantly higher for the improved variant ($\|\bar{\mathbf{e}}\|_2 = 0.35$). Why does the decoupled MsFV-method fail? The increased locality which comes from solving each edge problem independently seems to fail when a dual cell intersects both the shale layers and the rocks in between: This is especially apparent when looking at the small segment which is isolated near the lower left corner in the plots. In the original formulation, the error is low there, but with the decoupled formulation, the error is very high. Since both versions seem to struggle with the low porosity shale layers, this is not a big surprise. Care should therefore be taken when orienting dual grids for real world cases.



(a) Permeability (b) Permeability under shale layer

Figure 7.13: The geometry and geology of the Johansen formation

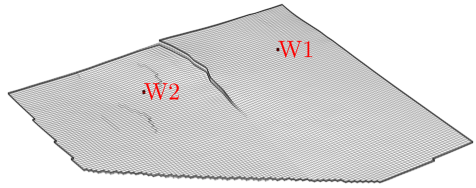
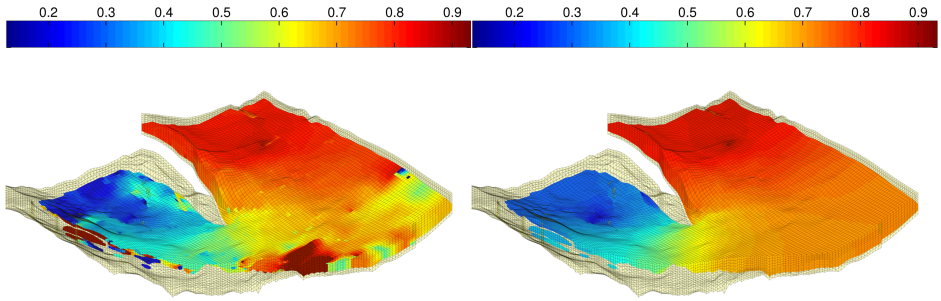
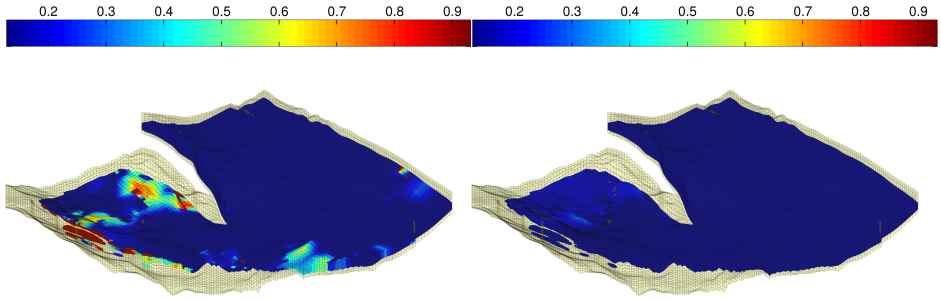


Figure 7.14: The well setup for the Johansen formation

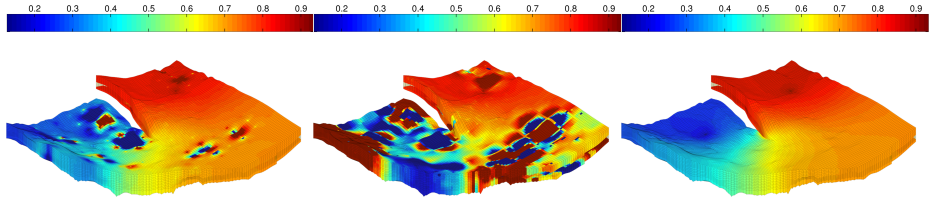


(a) MsFV-method (speedup) for the inner rocks (b) MsFV-method for the inner rocks



(c) Error (speedup) (d) Error

Figure 7.15: The results for the two methods for the inner rocks



(a) MsFV (b) MsFV (Speedup) (c) TPFA Reference

Figure 7.16: Results for the outer layers with very low permeability

7.6.3 SAIGUP

Introduction

For a second example of real world geometry, we will look at a dataset from the Sensitivity Analysis of the Impact of Geological Uncertainties on Production (SAIGUP) project. The project aims to estimate how geological uncertainties impact production forecasts, and has a publicly available dataset which can be used by MRST[Reg]. The dataset contains faults and complex geometry, with real world permeability characteristics.

Characteristics

The permeability can be seen in Figures 7.17a (horizontal) and 7.17b (vertical). The permeability of the SAIGUP model is fairly challenging, with both layers with different permeability as well as some channel like structures near the leftmost edge. The span in permeability is 10^7, which is almost as much as the challenging SPE10 cases, but the permeability values are in general not so close to zero as was the case in SPE10.

There are a number of faults all over the model which presents a more challenging geometry than the Johansen data set. Especially challenging is the faults which give a drop in height, but does not fully disconnect parts of the grid such as the fault in the Johansen formation. While we in Section 7.5.1 could adjust our coarse grid size to the fault location, the faults are here in many different locations with no obvious pattern. The partitioning algorithm employed is not specifically fault aware, but will hopefully be able to create a dual grid.

Methodology

The model is embedded in a logically Cartesian grid with a size of $40 \times 120 \times 20$ fine cells. Not all cells are marked as active, giving a total of 78720 cells to solve for. Some initial testing was done using homogeneous permeability to find an appropriate coarse grid: It was decided to be $13 \times 25 \times 3$ giving coarse blocks consisting of approximately $3 \times 5 \times 7 = 105$ fine cells. The testing using homogeneous permeability was done so that we can distinguish between the weaknesses of the *method* and the *partitioning algorithm*. As the MsFV-method has had problems with highly anisotropic mediums when solving subsets of SPE10, this is an important distinction. The small coarse block sizes ensure

that the coarse blocks are convex as well as fairly small. The z-direction is the coarsest partition because of the significant curvature in the domain - when there is a large amount of local curvature, the geometric approach of shooting planes can quickly categorize too many cells as edge cells.

The boundary condition is a large pressure differential from the right to the left of the model[2]. The pressure is 500 Pascal at the right and 200 Pascal at the left, with no-flow on the other faces. The boundary condition was chosen because it would ensure flow across the entire domain, showing any area where the method fails. Some areas could potentially also be isolated when partitioning the domain, which would be easy to see in a flow-everywhere scenario.

Results

Since the geometry and permeability is quite complex, we will plot both the top and bottom layer as well as a layer logically in the middle along the z-axis. The top and bottom is interesting because that is where the grid ends and the middle is interesting because it would be the most relevant for a real world simulation since most of the domain by definition would be the interior.

For the original formulation, the results are excellent: For the middle layer, shown in Figure 7.19b the solution is visually indistinguishable from the TPFA reference solution in Figure 7.19a. The top and bottom layers (Figures 7.18b and 7.20b respectively) are also very close to the TPFA solutions (shown in Figures 7.18a and 7.20a) with the exception of some small areas where the solution is fairly bad. These small areas correspond to primal coarse blocks where most of the logically defined block has been set to inactive, which results in too few cells to define a meaningful dual grid. An idea for further work would be to merge all coarse blocks under a certain size into the neighbours while still aiming for hexahedral coarse blocks.

The improved faster solution, however, fails dramatically when trying to solve SAIGUP. The results for the top, middle and bottom layer is shown in Figures 7.18c, 7.19c and 7.20c. The solution is unusable and bears little resemblance, if any, to the reference solutions. This could be because the dual grid is not oriented to the large changes in permeability - since the variant solves each edge problem independently, this could lead to singular submatrices which would be unproblematic in the original formulation, where there are multiple

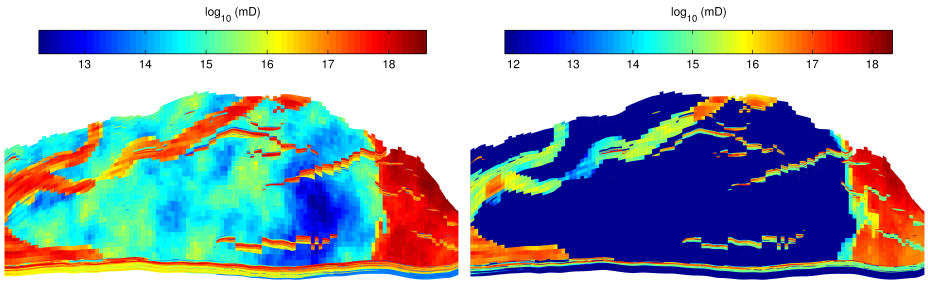
[2]The orientation left/right is used in the natural manner for the figures used here. In the model, left is the maximum y-coordinate and right the minimum y-coordinate.

Table 7.4: Error and runtime for the SAIGUP dataset

	Normal	Improved
$\|\bar{\mathbf{e}}\|_2$	0.147	191.771
$\|\bar{\mathbf{e}}\|_\infty$	13.652	766.245
$t_{operators}$	24.6s	19.46s

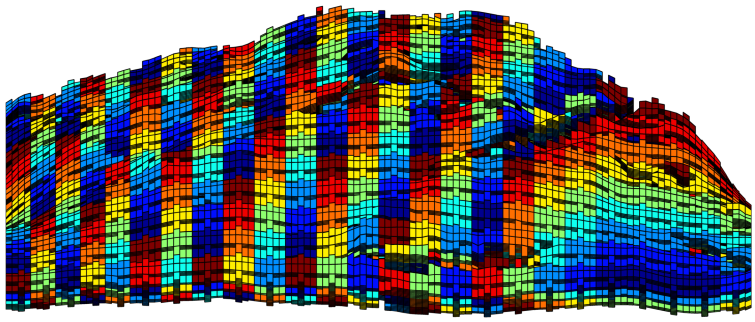
paths in the graph to each node.

The error and timing results can be seen in Table 7.4.



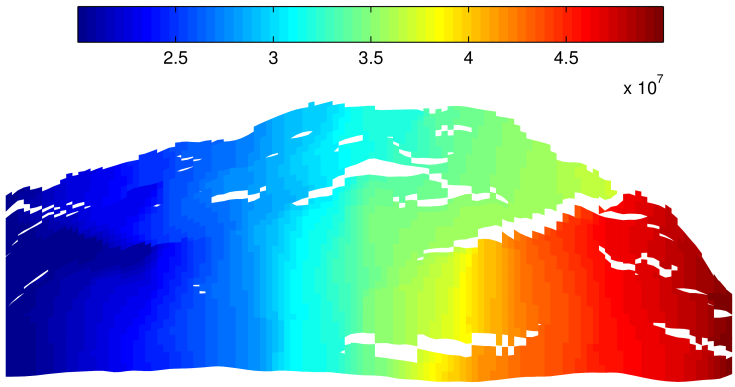
(a) Horizontal permeability

(b) Vertical permeability

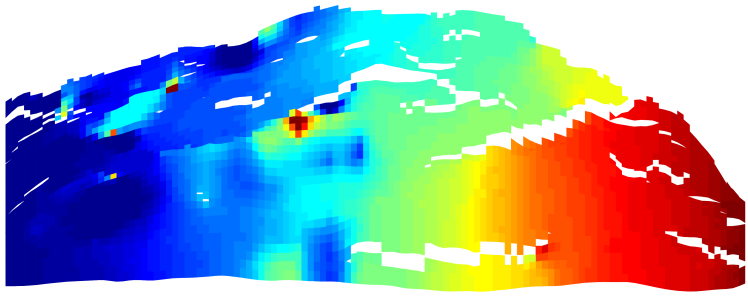


(c) Coarse grids

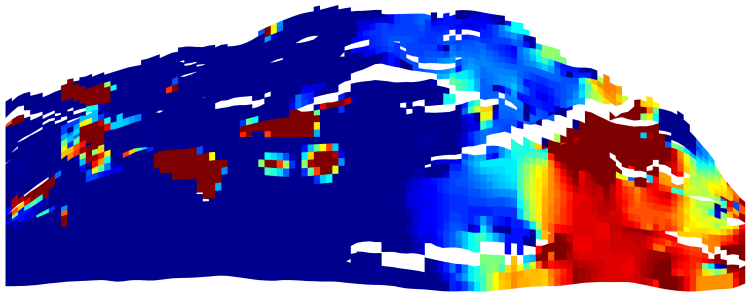
Figure 7.17: SAIGUP permeability set up, geometry and coarse grids



(a) Layer 1 Reference TPFA

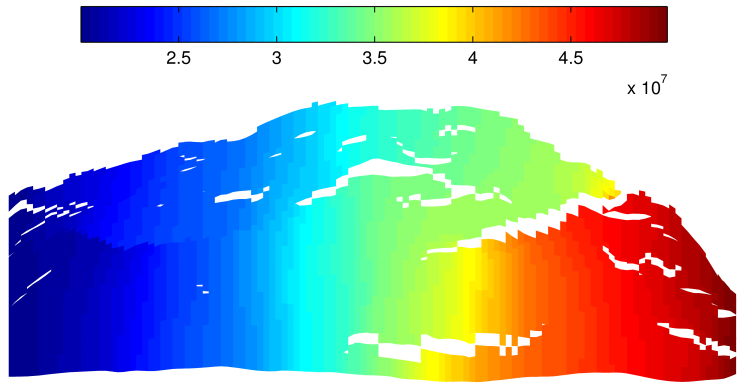


(b) Layer 1 MsFV-method, original formulation

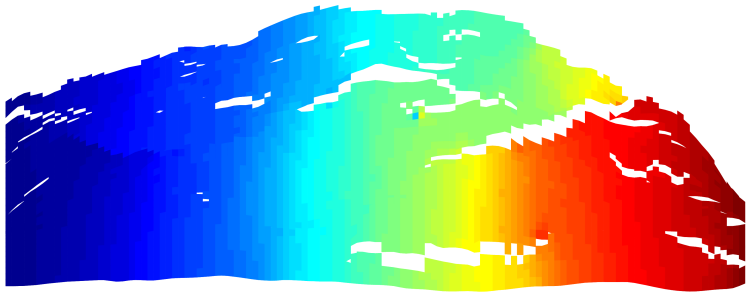


(c) Layer 1 MsFV-method, faster formulation

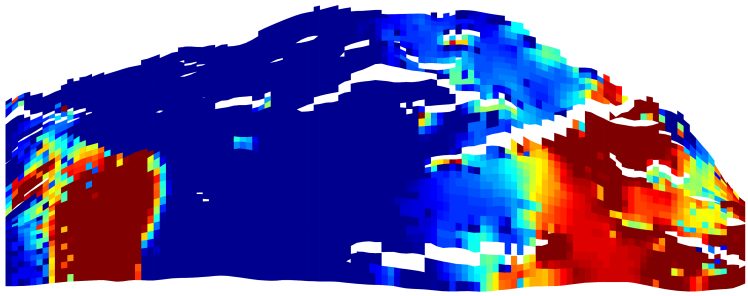
Figure 7.18: The lower layer of the SAIGUP case



(a) Layer 10 Reference TPFA

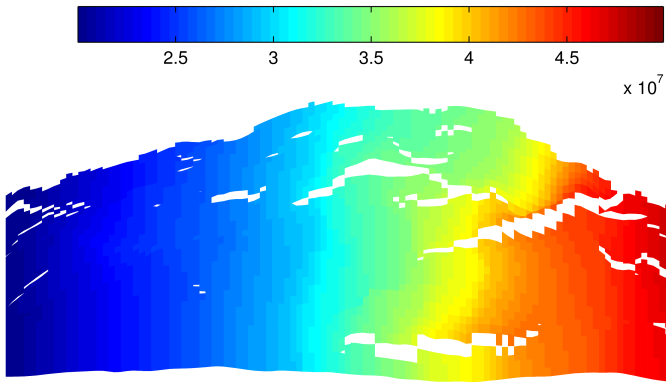


(b) Layer 10 MsFV-method, original formulation

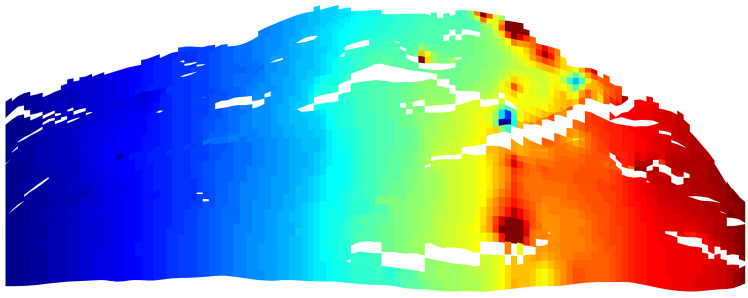


(c) Layer 10 MsFV-method, faster formulation

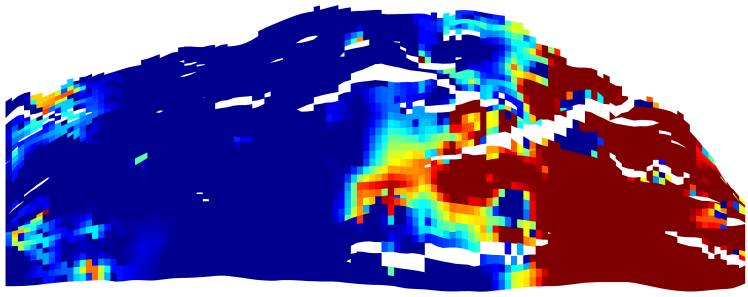
Figure 7.19: The middle layer of the SAIGUP case



(a) Layer 20 Reference TPFA



(b) Layer 20 MsFV-method, original formulation



(c) Layer 20 MsFV-method, faster formulation

Figure 7.20: The topmost layer of the SAIGUP case

7.7 Iterative solvers

When validating the iterative solvers, we will use the preconditioned residual,

$$\|M^{-1}Q(\tilde{r} - \tilde{A}\tilde{u})\|_2, \tag{7.6}$$

to measure the error. The preconditioner is applied to the residual to achieve parity between MATLAB's builtin GMRES implementation and the Dirichlet smoothers.

7.7.1 Flow channel

To test the implementation of the iterative solvers, we will first reuse the flow channel example from Section 7.3.1.

GMRES converges to machine precision very quickly, as seen in Figure 7.21a. The smoother cycles take much longer to converge, as shown in Figures 7.22a and 7.22b. Only 1000 iterations are shown, but the patterns are clear: Using smoothing cycles results in a steady drop during each cycle, with an small increase in error when doing another multiscale iteration. The effect of smoothing steps drop off after time, making another multiscale iteration necessary. This could theoretically be done adaptively, by for example measuring maximum of the residual each iteration before deciding to do more smoothing steps.

Since GMRES is obviously a lot faster, does this mean that smooth-cycles are useless? No, because GMRES is a complicated algorithm which requires global information during solving steps. It uses a lot of memory and is not easily suited for parallel processing in the way smoothers are: Since smoothers can work on each local coarse block, an implementation will have little communication overhead.

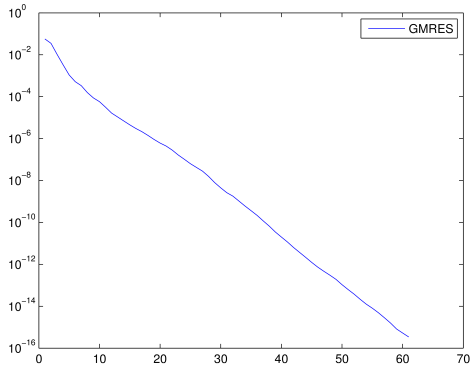
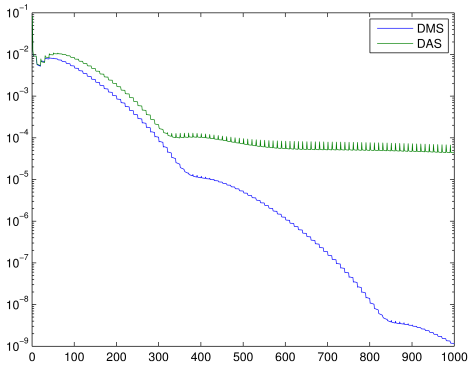
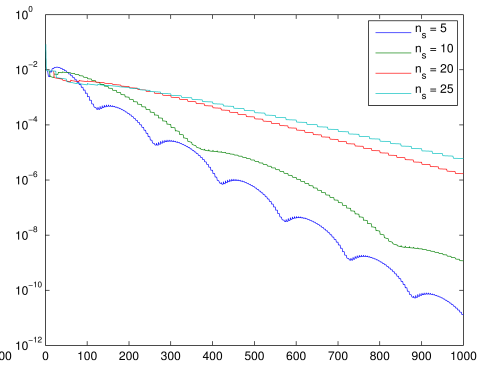


Figure 7.21: Convergence for GMRES applied to the flow channel example



(a) DAS and DMS iterations - $n_s = 10$



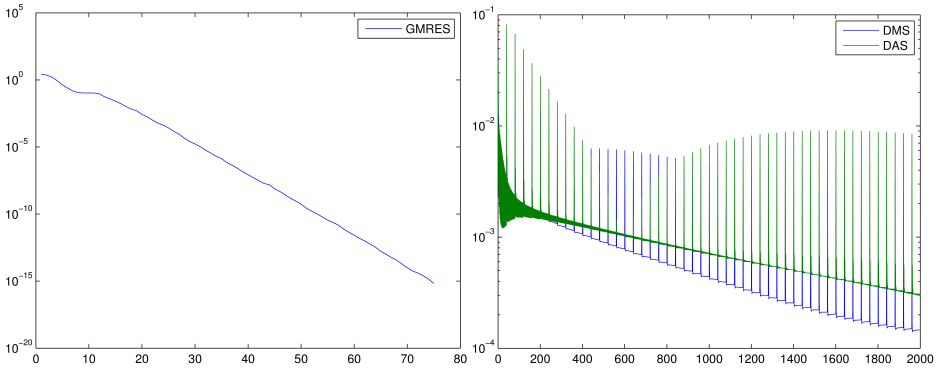
(b) Convergence for different smoothings steps n_s

Figure 7.22: A comparison of different smoothing cycles for the MsFV-iterations.

7.7.2 SAIGUP

We then turn our attention to the realistic SAIGUP case previously seen in Section 7.6.3. The GMRES and DAS/DMS-cycles were performed with a maximum of 2000 iterations. Each smooth cycle consisted of 40 steps. The original formulation of the MsFV was employed, as the faster variant diverged when iterated.

GMRES using the MsFV-method as a preconditioner, seen in Figure 7.23a, converges in 74 iterations. Doing GMRES without a MsFV preconditioner on the same initial residual does not results in convergence in 2000 iterations, which makes the use of the MsFV preconditioner worthwhile in this case.



(a) Preconditioned GMRES (b) DAS and DMS

Figure 7.23: The iterative results for the SAIGUP dataset.

For the smoother cycles, something interesting happens: As can be seen in Figure 7.23b, the residual oscillates wildly. If we study the graph more closely, however, it becomes apparent that the error increases after each MsFV-solution, but steadily converges when looking at the smoothing cycles. The reason for this is that in small part of the domain the grid seems fairly disconnected from the neighbours: This leads to the MsFV-method getting high error there regardless of the error in the other areas. This can be seen in Figure 7.24a, where the error in the lower layer is shown. The solution is converging in all areas except for this small area where the method fails, and the residual stagnates, dominating the norm. It should be noted that the iterative algorithm takes a very long time to converge, and any benefit to be had in real world scenarios would be to remove oscillatory errors with a select few iterations instead of doing a full

solve.

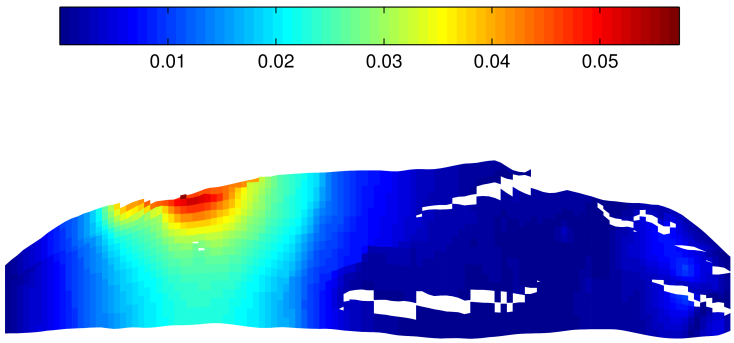


Figure 7.24: The error in the lower layer of the SAIGUP formation after 2000 DMS iterations.

7.8 Speed tests

7.8.1 Original formulation versus new formulation

The original formulation, as mentioned in Section REF, has some problems when extended to three dimensions because of the edge overlap in the linear systems. A new, analgous method, was formulated to solve this problem. To test the speed of the new methods, a benchmark was run where the computation of the operators, which includes all the pressure basis functions, was timed for different subsets of SPE10. Twelve runs were completed as shown in Figure 7.25a, which shows a significant speed increase for the new method. The first run takes the lower 5 layers of SPE10, the second the lower 10 and so on.

The benchmark was run on a computational server with 24 Intel Xeon 2.67GHz processors. When tested in serial mode, MATLAB was set to `-singleCompThread`. The speed observed in serial mode was similar to workstation performance, but the benchmarks were still run on the computational server to ensure a homogenous testing environment.

The speed increase is very significant. For the serial version, the speedup for the larger data sets lie between 6 and 7, giving the new method a significant advantage even with no parallelism. For the parallel version, the speedup was just below 5. Further parallelism could be extracted by exploiting the matrix structure described in Section REF by for example finding graph cycles using some graph algorithm [3] and reordering the systems accordingly. Note that MATLAB's built in linear algebra functionality is tuned for multiprocessing, and because of this the lower speedup for the 24 CPU case could simply be because a much larger dataset would be required for the runtime to be dominated by the linear systems.

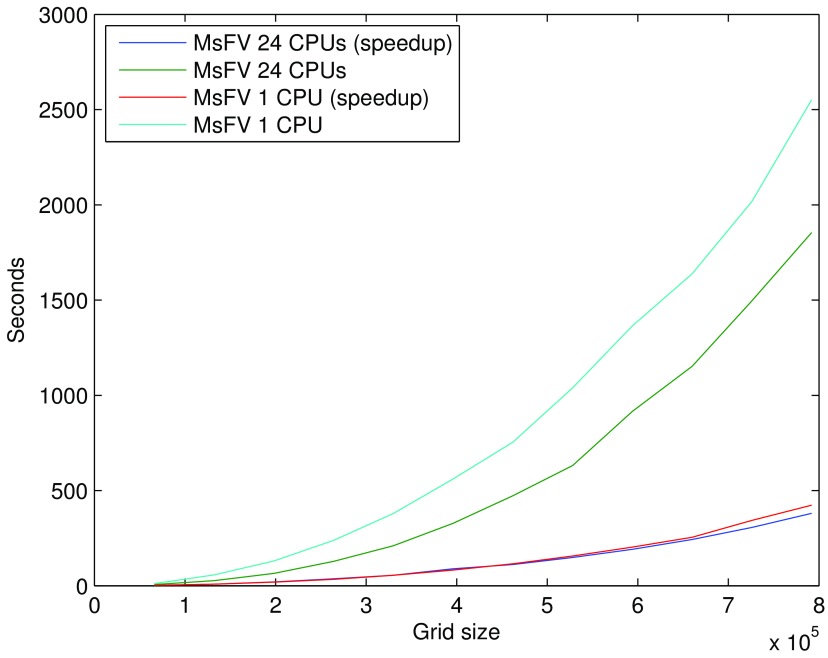
The error is also lower for the faster method by a factor 3 which is very promising, just like in Section 7.6.1.

7.8.2 Performance

Note: We will only concern ourselves with the improved method for this section.

The test setup is the same as in Section 7.8.1. The distribution of the time spent

[3] For example depth first search



(a) MsFV

Figure 7.25: The two variants of the method tested for both for a single CPU and for 24 CPUs.

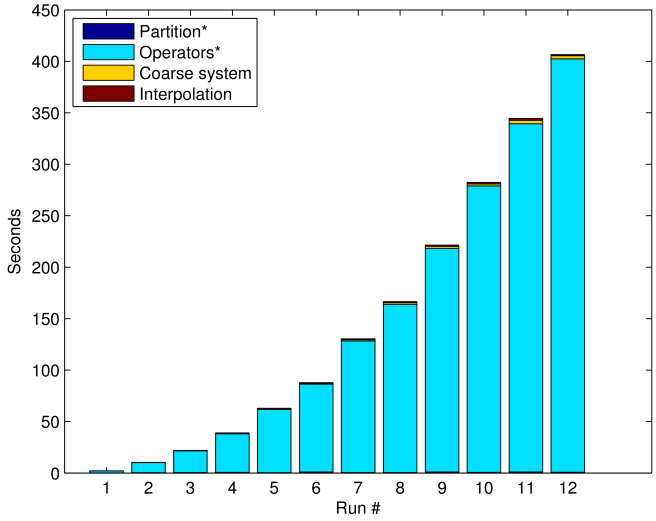
for the MsFV-method can be shown in Figure 7.26a. Note that the most computationally intensive steps are the creation of the dual partition and the operator generation. This is unsurprising, as the dual partition is computationally demanding and the operator generation is where all the large linear equation sets are solved. The operator generation seems to dominate the other steps as the problem size increases: Not unsurprising, as the partitioning algorithm was proposed to have a run time of $O(3N)$ where N is the number of fine cells. Solving linear equations is much more computationally intensive, even with the optimizations we have done.

The coarse system and the interpolation steps are dwarfed by the partitioning and operator generation. Fortunately, both partitioning and creating the operators can be considered to be *preprocessing steps* because they need only be done

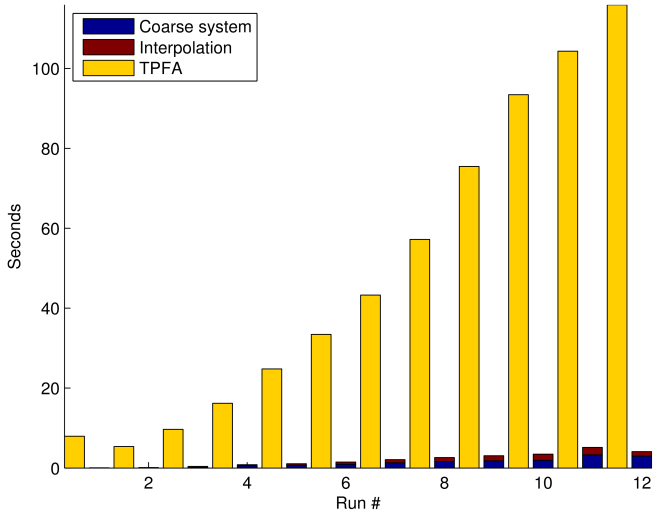
once for different combinations of boundary conditions and well setups [4]. The main appeal of the MsFV-method is in repeated solutions, for example when testing different well conditions or iterating a multiphase problem, and parallel computing.

The solution of the coarse system as well as the interpolation steps are however required every time a new solution is required. In Figure 7.26b, these steps are compared to the solution of the full TPFA system. As should be expected, solving the full system is vastly more expensive.

[4]Provided the wells are BHP wells which does not add equations to the system



(a) The time spent in different steps in the MsFV solver. Preprocessing steps marked with asterisk (*). Note that the solution time is vastly dominated by the operator generation, which indicates good implementation performance



(b) TPFA vs MsFV

Figure 7.26: The distribution of time spent for the different benchmarks. Solving a coarse system and interpolating is much cheaper than solving a full system.

Chapter 8

Conclusion

8.1 Summary of results

After a literature study an implementation of the Multiscale Finite Volume method was written for the MATLAB Reservoir Simulation Toolbox. The implementation handles single phase Darcy flow with boundary conditions, wells, flux reconstruction for conservative flow as well as three different approaches to iterative multiscale solutions (GMRES, DAS and DMS). The implementation has a focus on modularity and should be easy to modify for new experiments. The performance of the implementation after optimizations seems asymptotically dominated by the solution of the linear equations.

Several different methods for generating dual grids on unstructured grids were created and implemented, which handles Cartesian grids and delivers very promising results for corner point and unstructured grids. Altogether this represents the first implementation of the Multiscale Finite Volume method on complex 3D grids capable of handling fully unstructured grids, given some restrictions on the primal coarse grid. Other research on the subject includes work on logically Cartesian grids [SNA11] as well as different methods for handling faults [HDJ11].

A new variant of the MsFV-method was developed to overcome challenges in solution speed in 3D. The new variant is significantly faster and performs better on some datasets (SPE10: Tarbert and Ness) and worse on others (SAIGUP).

The new variant generates matrix systems which have greatly reduced condition numbers, and are easy to solve in parallel. Some analysis on parallelizing the operator formulation of the MsFV-method was developed, showing that the operator formulation with modifications can be solved using distributed computing. While the error is much larger on some complex datasets, experiments indicate that the large error is a result of both complex permeability and complex geometry: Using the same geometry with uniform permeability leads to low error, and the permeability is not more difficult than the upper Ness layers.

The method was validated on both 2D and 3D cases, including both synthetic and realistic datasets. While the method itself has problems with highly anisotropic mediums, the implementation itself does not seem like the limiting factor in any of the datasets attempted. With the exception of the highly challenging cases, the MsFV-methods error seems to be near around 1/10 in the relative norm. Using the method as both a preconditioner for GMRES and in smoother cycles, the error could be reduced to machine precision.

8.2 Further work

8.2.1 Partitioning algorithms

Creating dual grids is a very difficult problem, and there is always room for improvement. Adaptive coarsening of the grid with dynamic adjustments of the coarse grid based on local changes in permeability is one possible idea to handle the weaknesses of the method. The other is to both improve the speed and success rate of the dual grid generation, for example by combining logical and geometric partitioning so that the logical indices is used in areas without faults. The dual grid generation could also be extended to grids where hexahedral coarse blocks does not make sense, for instance in 2.5D PEBI grids.

The primal partitioning algorithms are not especially geometry aware, which forces the user to make repeated attempts to find an coarse primal grid which results in a suitable dual grid for the MsFV-method. An improved algorithm could postprocess the grids created by MRST's dual grid functionality to give better results with the method.

For grids with special orientations, such as PEBI grids which have six sides in each cell in the xy-plane, a logically Cartesian grid with the same bounding coordinates could be produced. Since the dual grid is trivial to produce for

such a grid, this could then be used to sample the nearest cells in the PEBI grid to categorize the nodes. This would probably require some postprocessing to ensure that the edge cells are connected.

8.2.2 Parallel implementation

The MsFV-method is very well suited for parallel implementation and implementing a distributed solver in a more low level programming language would be a natural next step for anyone interested in using the MsFV-method for real world cases with more than a few million fine cells.

If one has access to the parallel computing toolbox, many parts of the MsFV-module can be parallelized with little or no changes to the underlying code, for example the dual grid generation. Unfortunately, I did not have access to this toolbox on the university workstations where I wrote most of the code. Note that the solver is still parallel in the most computationally intensive parts (basis generation and interpolation) where the parallelism depends on the implementation of mldivide in MATLAB. A special linear solver which can detect the decoupled problems and solve them independently could possibly make the implementation faster, and make a distributed solve with minimized communication costs.

8.2.3 Other discretizations

While the implementation currently uses the TPFA discretization, creating a solver using other discretizations should be fairly straight forward, depending on the matrix systems involved and could significantly improve the accuracy of the solutions. For example, implementing multipoint stencils in the current framework would only require an categorization of the nodes corresponding to boundary conditions into the familiar inner, edge and central node categories.

Appendix A

Rapid experimentation using setup GUI

The MSFV-method has a large amount of possible configuration parameters:

- Different geometries have different challenges
- Some types permeabilities can have an huge qualitative influence on the solution
- The choice of coarse partitions must be done with care, as the method is very sensitive to the choice of partitioning around complex geometry
- With the different dual partitioning schemes developed in this thesis, there are several options which all influence the error
- Boundary conditions and wells obviously have a big influence on the solution

Clearly this many degrees of freedom can be intimidating when trying to understand the advantages and disadvantages of an implementation. Early in the process, a decision was made to set up a user interface to make it easy to experiment with different parameters and problems. This would not only make it

easier to test the method systematically, but it would also facilitate experimentation, which is a key component in scientific work. Several of the cases used in Chapter 7 were selected after experiments with the user interface.

The user interface itself can be seen as a wrapper for the MSFVM_Testing function, which is a function containing set up procedures for a wide range of grids, sizes, permeabilities and other configurations. Since grid selection is distinct from say boundary conditions, this functions offers a wide range of cases meant to encompass many aspects of the testing of reservoir simulation methods, not necessarily restricted to the MsFV-method. The interface also makes it trivial to reproduce many of the results in this thesis if further analysis is needed. A screenshot can be seen in Figure A.1.

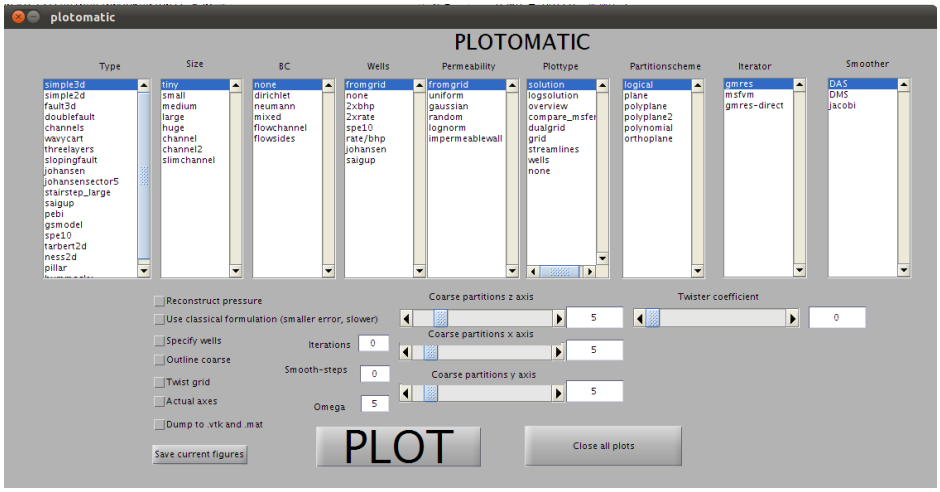


Figure A.1: The setup GUI

Bibliography

[Aav04] I. Aavatsmark, *Bevarelsesmetoder for elliptiske diferensialligninger*.

[CB01] M.A. Christie and M.J. Blunt, *Tenth spe comparative solution project: A comparison of upscaling techniques*, SPE Reservoir Evaluation & Engineering **4** (2001), no. 4, 308–317.

[EDH+09] G.T. Eigestad, H.K. Dahle, B. Hellevang, F. Riis, W.T. Johansen, and E. Øian, *Geological modeling and simulation of co 2 injection in the johansen formation*, Computational Geosciences **13** (2009), no. 4, 435–450.

[HBHJ08] H. Hajibeygi, G. Bonfigli, M.A. Hesse, and P. Jenny, *Iterative multiscale finite-volume method*, Journal of Computational Physics **227** (2008), no. 19, 8604–8621.

[HDJ11] H. Hajibeygi, R. Deb, and P. Jenny, *Multiscale finite volume method for non-conformal coarse grids arising from faulted porous media*, SPE Reservoir Simulation Symposium, 2011.

[HJ09] H. Hajibeygi and P. Jenny, *Multiscale finite-volume method for parabolic problems arising from compressible multiphase flow in porous media*, Journal of Computational Physics **228** (2009), no. 14, 5129–5147.

[HJ11] ———, *Adaptive iterative multiscale finite volume method*, Journal of Computational Physics **230** (2011), no. 3, 628–643.

[HW97] Thomas Y. Hou and Xiao-Hui Wu, *A multiscale finite element method for elliptic problems in composite materials and porous media*, Journal of Computational Physics **134** (1997), no. 1, 169–189.

[JLT03] P Jenny, S.H Lee, and H.A Tchelepi, *Multi-scale finite-volume method for elliptic problems in subsurface flow simulation*, Journal of Computational Physics **187** (2003), no. 1, 47–67.

[JLT05] P. Jenny, SH Lee, and HA Tchelepi, *Adaptive multiscale finite-volume method for multiphase flow and transport in porous media*, Multiscale Modeling and Simulation **3** (2005), no. 1, 50–64.

[KAL08] V. Kippe, J.E. Aarnes, and K.A. Lie, *A comparison of multiscale methods for elliptic problems in porous media flow*, Computational Geosciences **12** (2008), no. 3, 377–398.

[LAKK] K.A. Lie, J.E. Aarnes, V. Kippe, and S. Krogstad, *Multiscale methods for flow in porous media*.

[LeV07] R.J. LeVeque, *Finite difference methods for ordinary and partial differential equations*, Society for Industrial and Applied Mathematics Philadelphia, 2007.

[LJ07] I. Lunati and P. Jenny, *Treating highly anisotropic subsurface flow with the multiscale finite-volume method*, Multiscale Modeling & Simulation **6** (2007), 308.

[LTL10] I. Lunati, M. Tyagi, and S.H. Lee, *An iterative multiscale finite volume algorithm converging to the exact solution*, Journal of Computational Physics (2010).

[NB08] JM Nordbotten and PE Bjørstad, *On the relationship between the multiscale finite-volume method and domain decomposition preconditioners*, Computational Geosciences **12** (2008), no. 3, 367–376.

[Pea83] D.W. Peaceman, *Interpretation of well-block pressures in numerical reservoir simulation with nonsquare grid blocks and anisotropic permeability*, Old SPE Journal **23** (1983), no. 3, 531–543.

[Reg] Norsk Regnesentral, *The saigup project*, `http://www.nr.no/pages/sand/area_res_char_saigup`.

[Saa03] Y. Saad, *Iterative methods for sparse linear systems*, Society for Industrial Mathematics, 2003.

[SNA11] A. Sandvin, J.M. Nordbotten, and I. Aavatsmark, *Multiscale mass conservative domain decomposition preconditioners for elliptic problems on irregular grids*, Computational Geosciences **15** (2011), no. 3, 587–602.

[ZT08] H. Zhou and H. Tchelepi, *Operator-based multiscale method for compressible flow*, SPE Journal **13** (2008), no. 2, 267–273.