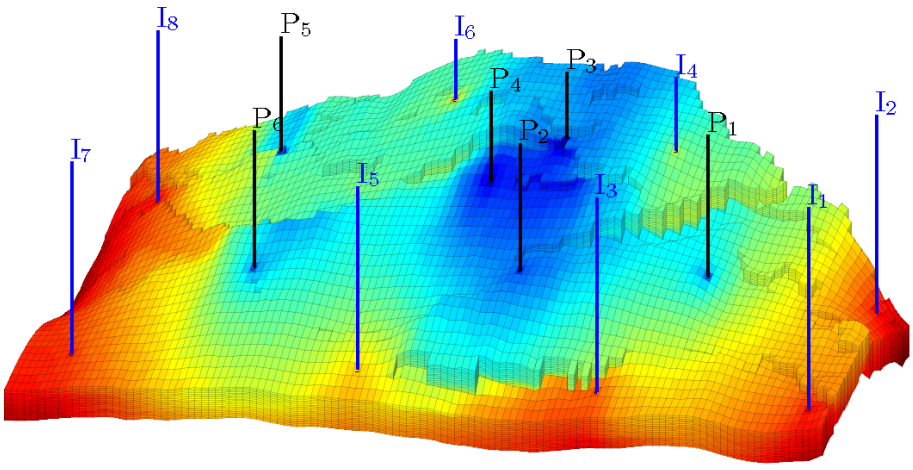
Knut-Andreas Lie

An Introduction to Reservoir Simulation Using MATLAB

User Guide for the Matlab Reservoir Simulation Toolbox (MRST)

May 27, 2014



SINTEF ICT, Departement of Applied Mathematics
Oslo, Norway

Preface

There are many good books that describe mathematical models for flow in porous media and present numerical methods that can be used to discretize and solve the corresponding systems of partial differential equations. However, neither of these books describe how to do this in practice. Some may present algorithms and data structures, but most leave it up to you to figure out all the nitty-gritty details you need to get your implementation up and running. Likewise, you may read papers that present models or computational methods that may be exactly what you need for your work. After the first enthusiasm, however, you very often end up quite disappointed–or at least, I do–when I realize that the authors have not presented all the details of their model, or that it will probably take me months to get my own implementation working.

In this book, I try to be a bit different and give a reasonably self-contained introduction to the simulation of flow and transport in porous media that also discusses how to implement the models and algorithms in a robust and efficient manner. In the presentation, I have tried to let the discussion of models and numerical methods go hand in hand with numerical examples that come fully equipped with codes and data, so that you can rerun and reproduce the results by yourself and use them as a starting point for your own research and experiments. All examples in the book are based on the Matlab Reservoir Simulation Toolbox (MRST), which has been developed by my group and published online as free open-source code under the GNU General Public License since 2009.

The book can alternatively be seen as a comprehensive user-guide to MRST. Over the years, MRST has become surprisingly popular (the latest releases have more than a thousand unique downloads each) and has expanded rapidly with new features. Unfortunately, the manuscript has not been able to keep pace. The current version is up-to-date with respect to the latest development in data structures and syntax, but only includes material on single-phase flow. However, more material is being added almost every day, and the manuscript will hopefully be expanded to cover multiphase flow and various workflow tools and special-purpose solvers in the not too distant future.

I hereby grant you permission to use the manuscript and the accompanying example scripts for your own educational purpose, but please do not reuse or redistribute this material as a whole, or in parts, without explicit permission. Moreover, notice that *the current manuscript is a snapshot of work in progress and is far from complete*. The text may contain a number of misprints and errors, and I would be grateful if you help to improve the manuscript by sending me an email. Suggestions for other improvement are also most welcome.

Oslo, *Knut-Andreas Lie*
May 27, 2014 *Knut-Andreas.Lie@sintef.no*

Contents

1 Introduction 1
- 1.1 Reservoir Simulation 2
- 1.2 About This Book 4
- 1.3 The First Encounter with MRST 5
- 1.4 More about MRST 6
- 1.5 About examples and standard datasets 9

Part I Geological Models and Grids

2 Modelling Reservoir Rocks 15
- 2.1 Formation of a Sedimentary Reservoir 15
- 2.2 Multiscale Modelling of Permeable Rocks 17
  - 2.2.1 Macroscopic Models 19
  - 2.2.2 Representative Elementary Volumes 20
  - 2.2.3 Microscopic Models: The Pore Scale 21
  - 2.2.4 Mesoscopic Models 23
- 2.3 Modelling of Rock Properties 24
  - 2.3.1 Porosity 25
  - 2.3.2 Permeability 26
  - 2.3.3 Other parameters 28
- 2.4 Rock Modelling in MRST 29
  - 2.4.1 Homogeneous Models 29
  - 2.4.2 Random and Lognormal Models 30
  - 2.4.3 10^{th} SPE Comparative Solution Project: Model 2 31
  - 2.4.4 The Johansen Formation 34
  - 2.4.5 The SAIGUP Model 36

3 Grids in Subsurface Modeling 43
- 3.1 Structured Grids 45
- 3.2 Unstructured Grids 49

3.2.1 Delaunay Tessellation 50
3.2.2 Voronoi Diagrams 53
3.3 Stratigraphic Grids 55
3.3.1 Corner-Point Grids 56
3.3.2 Layered 2.5D PEBI Grids 66
3.4 Grid Structure in MRST 70

4 Grid Coarsening 87
4.1 Partition Vectors 88
4.1.1 Uniform Partitions 88
4.1.2 Connected Partitions 89
4.1.3 Composite Partitions 90
4.2 Coarse Grid Representation in MRST 93
4.2.1 Subdivision of Coarse Faces 94
4.3 Coarsening of Realistic Reservoir Models 97
4.3.1 The Johansen Aquifer 97
4.3.2 The Shallow-Marine SAIGUP Model 100
4.4 General Advice and Simple Guidelines 104

Part II Single-Phase Flow

5 Mathematical Models and Basic Discretizations 109
5.1 Fundamental concept: Darcy's law 109
5.2 General flow equations for single-phase flow 111
5.3 Auxiliary conditions and equations 115
5.3.1 Boundary and initial conditions 116
5.3.2 Models for injection and production wells 117
5.3.3 Field lines and time-of-flight 119
5.3.4 Tracers and volume partitions 120
5.4 Basic finite-volume discretizations 121
5.4.1 A two-point flux-approximation (TPFA) method 122
5.4.2 Abstract formulation: discrete `div` and `grad` operators . 126
5.4.3 Discretizing the time-of-flight and tracer equations 128

6 Incompressible Solvers in MRST 131
6.1 Basic data structures 131
6.1.1 Fluid properties 131
6.1.2 Reservoir states 132
6.1.3 Fluid sources 133
6.1.4 Boundary conditions 134
6.1.5 Wells 135
6.2 Incompressible two-point pressure solver 137
6.3 Upwind solver for time-of-flight and tracer 140
6.4 Simulation examples 143

6.4.1 Quarter-five spot . . . 143
6.4.2 Boundary conditions . . . 147
6.4.3 Structured versus unstructured stencils . . . 151
6.4.4 Using Peaceman well models . . . 156

7 Single-Phase Solvers Based on Automatic Differentiation . . 161
7.1 Implicit discretization . . . 161
7.2 Automatic differentiation . . . 163
7.3 Automatic differentiation in MRST . . . 164
7.4 An implicit single-phase solver . . . 168
7.4.1 Model setup and initial state . . . 168
7.4.2 Discrete operators and equations . . . 170
7.4.3 Well model . . . 171
7.4.4 The simulation loop . . . 172
7.5 Rapid prototyping . . . 175
7.5.1 Pressure-dependent viscosity . . . 175
7.5.2 Non-Newtonian fluid . . . 178

8 Consistent Discretizations on Polyhedral Grids . . . 185
8.1 The Mixed Finite-Element Method . . . 188
8.1.1 Continuous Formulation . . . 188
8.1.2 Discrete Formulation . . . 190
8.1.3 Hybrid formulation . . . 193
8.2 Consistent Methods on Mixed Hybrid Form . . . 195
8.3 The Mimetic Method . . . 198
8.3.1 General Family of Inner Products . . . 199
8.3.2 General Parametric Family . . . 202
8.3.3 Two-Point Type Methods . . . 202
8.3.4 Raviart–Thomas Type Inner Product . . . 204
8.3.5 Default Inner Product in MRST . . . 206
8.3.6 Local-Flux Mimetic Method . . . 206

References . . . 209

1

Introduction

Modelling of flow processes in the subsurface is important for many applications. In fact, subsurface flow phenomena cover some of the most important technological challenges of our time. The road toward sustainable use and management of the earth's groundwater reserves necessarily involves modelling of groundwater hydrological systems. In particular, modelling is used to acquire general knowledge of groundwater basins, quantify limits of sustainable use, monitor transport of pollutants in the subsurface, and appraise schemes for groundwater remediation.

A perhaps equally important problem is how to reduce emission of greenhouse gases, such as CO2, into the atmosphere. Carbon sequestration in porous media has been suggested as a possible means. The primary concern related to storage of CO2 in subsurface rock formations is how fast the stored CO2 will escape back to the atmosphere. Repositories do not need to store CO2 forever, just long enough to allow the natural carbon cycle to reduce the atmospheric CO2 to near pre-industrial level. Nevertheless, making a qualified estimate of the leakage rates from potential CO2 storage facilities is a nontrivial task, and demands interdisciplinary research and software based on state-of-the art numerical methods for modelling subsurface flow. Other questions of concern is whether the stored CO2 will leak into fresh-water aquifers or migrate to habitated or different legislative areas

The third challenge is petroleum production. The civilized world will (most likely) continue to depend on the utilization of petroleum resources both as an energy carrier and as a raw material for consumer products the next 30–40 years. Given the decline in conventional petroleum production and the reduced rate of new major discoveries, optimal utilization of current fields and discoveries is of utter importance to meet the demands for petroleum and lessen the pressure on exploration in vulnerable areas like in the arctic regions. Likewise, there is a strong need to understand how unconventional petroleum resources can be produced in an economic way that minimizes the harm to the environment.

Reliable computer modeling of subsurface flow is much needed to overcome these three challenges, but is also needed to exploit deep geothermal energy, ensure safe storage of nuclear waster, improve remediation technologies to remove contaminants from the subsurface, etc. Indeed, the need for tools that help us understand flow processes in the subsurface is probably greater than ever, and increasing. More than fifty years of prior research in this area has led to some degree of agreement in terms of how subsurface flow processes can be modelled adequately with numerical simulation technology. Most of our prior research in this area has targeted reservoir simulation, i.e., modelling flow in oil and gas reservoirs, and hence we will mainly focus on this application in this book. However, the general modelling framework, and the numerical methods that are discussed, apply also to modelling other types of flow in consolidated and saturated porous media.

Techniques developed for the study of subsurface flow are also applicable to other natural and man-made porous media such as soils, biological tissues and plants, filters, fuel cells, concrete, textiles, polymer composites, etc. A particular interesting example is in-tissue drug delivery, where the challenge is to minimize the volume swept by the injected fluid. This is the complete opposite of the challenge in petroleum production, in which one seeks to maximize the volumetric sweep of the injected fluid to push as much petroleum out as possible.

1.1 Reservoir Simulation

Reservoir simulation is the means by which we use a numerical model of the petrophysical characteristics of a hydrocarbon reservoir to analyze and predict fluid behavior in the reservoir over time. Simulation of petroleum reservoirs started in the mid 1950's and has become an important tool for qualitative and quantitative prediction of the flow of fluid phases. Reservoir simulation is a complement to field observations, pilot field and laboratory tests, well testing and analytical models and is used to estimate production characteristics, calibrate reservoir parameters, visualize reservoir flow patterns, etc. The main purpose of simulation is to provide an information database that can help the oil companies to position and manage wells and well trajectories to maximize the oil and gas recovery. Generally, the value of simulation studies depends on what kind of extra monetary or other profit they will lead to, e.g., by increasing the recovery from a given reservoir. However, even though reservoir simulation can be an invaluable tool to enhance oil-recovery, the demand for simulation studies depends on many factors. For instance, petroleum fields vary in size from small pockets of hydrocarbon that may be buried just a few meters beneath the surface of the earth and can easily be produced, to huge reservoirs stretching out several square kilometres beneath remote and stormy seas, for which extensive simulation studies are inevitable to avoid making incorrect, costly decisions.

To describe the subsurface flow processes mathematically, two types of models are needed. First, one needs a mathematical model that describes how fluids flow in a porous medium. These models are typically given as a set of partial differential equations describing the mass-conservation of fluid phases, accompanied by a suitable set of constitutive relations. Second, one needs a geological model that describes the given porous rock formation (the reservoir). The geological model is realized as a grid populated with petrophysical properties that are used as input to the flow model, and together they make up the reservoir simulation model.

Unfortunately, obtaining an accurate prediction of reservoir flow scenarios is a difficult task. One of the reasons is that we can never get a complete and accurate characterization of the rock parameters that influence the flow pattern. And even if we did, we would not be able to run simulations that exploit all available information, since this would require a tremendous amount of computer resources that exceed by far the capabilities of modern multiprocessor computers. On the other hand, we do not need, nor do we seek a simultaneous description of the flow scenario on all scales down to the pore scale. For reservoir management it is usually sufficient to describe the general trends in the reservoir flow pattern.

In the early days of the computer, reservoir simulation models were built from two-dimensional slices with 10^2–10^3 Cartesian grid cells representing the whole reservoir. In contrast, contemporary reservoir characterization methods can model the porous rock formations by the means of grid-blocks down to the meter scale. This gives three-dimensional models consisting of multi-million cells. Stratigraphic grid models, based on extrusion of 2D areal grids to form volumetric descriptions, have been popular for many years and are the current industry standard. However, more complex methods based on unstructured grids are gaining in popularity.

Despite an astonishing increase in computer power, and intensive research on computation techniques, commercial reservoir simulators can seldom run simulations directly on geological grid models. Instead, coarse grid models with grid-blocks that are typically ten to hundred times larger are built using some kind of upscaling of the geophysical parameters. How one should perform this upscaling is not trivial. In fact, upscaling has been, and probably still is, one of the most active research areas in the oil industry. This effort reflects that it is a general opinion that, with the ever increasing size and complexity of the geological reservoir models, one cannot generally expect to run simulations directly on geological models in the foreseeable future.

Along with the development of better computers, new and more robust upscaling techniques, and more detailed reservoir characterizations, there has also been an equally significant development in the area of numerical methods. State-of-the-art simulators employ numerical methods that can take advantage of multiple processors, distributed memory workstations, adaptive grid refinement strategies, and iterative techniques with linear complexity. For the simulation, there exists a wide variety of different numerical schemes that all

have their pros and cons. With all these techniques available we see a trend where methods are being tuned to a special set of applications, as opposed to traditional methods that were developed for a large class of differential equations.

1.2 About This Book

The book is intended to serve several purposes. First, we wish to give a self-contained introduction to the basic theory of flow in porous media and the numerical methods used to solve the underlying differential equations. In doing so, we will present both the basic model equations and physical parameters, classical numerical methods that are the current industry standard, as well as more recent methods that are still being researched by academia. The presentation of computational methods and modeling concepts is accompanied by illustrative examples ranging from idealized and highly simplified examples to real-life cases.

All visual and numerical examples presented in this book have been created using the MATLAB Reservoir Simulation Toolbox (MRST). This open-source toolboxes contains a comprehensive set of routines and data structures for reading, representing, processing, and visualizing structured and unstructured grids, with particular emphasis on the corner-point format used within the petroleum industry. The core part of the toolbox contains basic flow and transport solvers that can be used to simulate incompressible, single- and two-phase flow on general unstructured grids. Solvers for more complex flow physics as well as special-purpose computational tools for various workflows can be found in a large set of add-on modules. The second purpose of the book is therefore to give a presentation of the various functionality in MRST. By providing data and sufficient detail in all examples, we hope that the interested reader will be able to repeat and modify our examples on his/her own computer. We hope that this material can give other researchers, or students about to embark on a Master or a PhD project, a head start.

MRST was originally developed to support our research on consistent discretization and multiscale solvers on unstructured, polyhedral grids, but has over the years developed into an efficient platform for rapid prototyping and efficient testing of new mathematical models and simulation methods. A particular aim of MRST is to accelerate the process of moving from simplified and conceptual testing to validation of realistic setups. The book is therefore focused on questions that are relevant to the petroleum industry. However, to benefit readers that interested in developing more generic computational methodologies, we also try to teach a few general principles and methods that are useful for developing flexible and efficient MATLAB solvers for other applications of porous media flow or on unstructured polyhedral grids in general. In particular, we seek to provide sufficient implementation details so that the interested reader can use MRST as a starting point for his/her own develop-

ment. Being an open-source software, it is our hope that readers of this book can contribute to extend MRST in new directions.

Over the last few years, key parts of MRST have become relatively mature and well tested. This has enabled a stable release policy with two releases per year, typically in April and October. Throughout the releases, the basic functionality like grid structures has remained largely unchanged, except for occasional and inevitable bugfixes, and the primary focus has been on expanding functionality by maturing and releasing in-house prototype modules. However, MRST is mainly developed and maintained as an efficient prototyping tool to support contract research carried out by SINTEF for the energy-resource industry and public research agencies. Fundamental changes will therefore occur from time to time, e.g., like when automatic differentiation was introduced in 2012. In writing this, we (regretfully) acknowledge the fact that specific code details (and examples) in books that describe evolving software tend to become somewhat outdated. To countermand this, we intend to keep an up-to-date version of all examples in the book on the MRST webpage:

http://www.sintef.no/Projectweb/MRST/

These examples are designed using cell-mode scripts, which can be seen as a type of "MATLAB workbook" that allowes you break the scripts down into smaller pieces that can be run individually to perform a specific subtask such as creating a parts of a model or making an illustrative plot. In our opinion, the best way to understand the examples is to go through the script, evaluating one cell at at time. Alternatively, you can set a breakpoint on the first line, and step through the script in debug mode. Some of the scripts contain more text and are designed to make easily published documents. If you are not familiar with cell-mode scripts, or debug mode, we strongly urge you to learn these useful features in MATLAB as soon as possible.

You are now ready to start digging into the material. However, before doing so, we present an example that will give you a first taste of flow in porous media and the MATLAB Reservoir Simulation Toolbox.

1.3 The First Encounter with MRST

The purpose of the example is to show the basic steps for setting up, solving, and visualizing a simple flow problem. To this end, we will compute a known analytical solution: the linear pressure solution describing hydrostatic equilibrium for an incompressible, single-phase fluid. The basic model in subsurface flow consists of an equation expressing conservation of mass and a constitutive relation called Darcy's law that relates the volumetric flow rate to the gradient of flow potential

$$\nabla \cdot \vec{v} = 0, \qquad \vec{v} = - - \frac{K}{\mu}\big[\nabla p + \rho g \nabla z\big], \tag{1.1}$$

where the unknowns are the pressure p and the flow velocity $\vec{v}$. By eliminating $\vec{v}$, we can reduce (1.1) to the elliptic Poisson equation. In (1.1), the rock is characterized by the permeability K that gives the rock's ability to transmit fluid. Here, K is set to 100 milli-darcies (mD). The fluid has a density ρ of 1000 kg/m^3 and viscosity μ equal 1 cP, g is the gravity constant, and z is the depth. More details on these flow equations, the rock and fluid parameters, the computational method, and its MATLAB implementation will be given throughout the book.

The computational domain is a square column, $[0,1]\times[0,1]\times[0,30]$, which we discretize using a regular $1\times1\times30$ Cartesian grid. To close (1.1), we must describe conditions on all boundaries. To this end, we prescribe $p = 100$ bar at the top of the column and no-flow conditions ($\vec{v}\cdot n = 0$) elsewhere. The simulation model is set up by constructing a grid, setting the rock permeability, instantiating a fluid object, and setting boundary conditions:

```
gravity reset on
G        = cartGrid([1, 1, 30], [1, 1, 30]);
G        = computeGeometry(G);
rock.perm = repmat(0.1*darcy(), [G.cells.num, 1]);
fluid    = initSingleFluid('mu' ,  1*centi*poise, ...
                           'rho', 1014*kilogram/meter^3);
bc       = pside([], G, 'TOP', 100.*barsa());
```

MRST works in SI units, and hence we must be careful to specify the correct units for all physical quantities.

To solve (1.1), we will use a standard two-point finite-volume scheme (which here coincides with the classical seven-point scheme for Poisson's problem). First, we compute the matrix coefficient, which are called transmissibilities, and then use these to assemble and solve the discrete system

```
T = computeTrans(G, rock);
sol = incompTPFA(initResSol(G, 0.0), G, T, fluid,'bc', bc);
```

Having computed the solution, we plot the pressure given in unit 'bar':

```
plotFaces(G, 1:G.faces.num, convertTo(sol.facePressure, barsa()));
set(gca, 'ZDir', 'reverse'), title('Pressure [bar]')
view(3), colorbar, set(gca,'DataAspect',[1 1 10])
```

The result is shown in Figure 1.1.

1.4 More about MRST

MRST is organized quite similar to MATLAB and consists of a collection of core routines and a set of add-on modules. The core consists of routines and data structures for creating and manipulating grids and physical properties, utilities for performing automatic differentiation (you write the formulas and

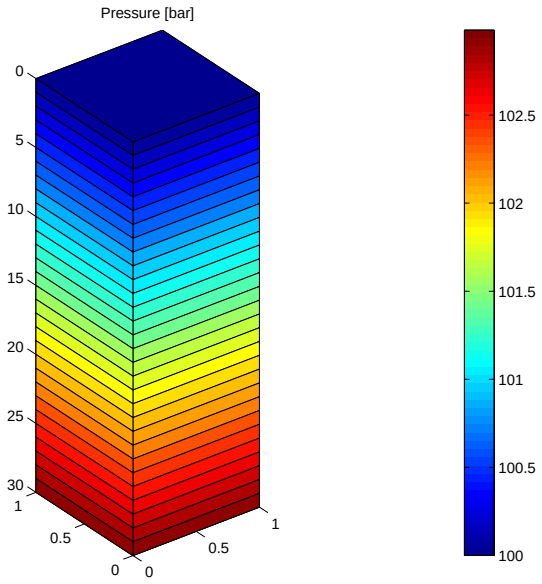


Fig. 1.1. Hydrostatic pressure distribution in a gravity column computed by MRST. This is example is taken from the MRST tutorial `gravityColumn.m`

specify the independent variables, MRST computes the corresponding Jacobians), as well as a few routines for plotting cell and face data defined over a grid. In addition, the core contains a few basic solvers for incompressible, immiscible, single-phase and two-phase flow. The functionality in MRST core is considered to be stable and not expected to change (significantly) in future releases. The introductory parts of the book will rely entirely on routines from MRST core.

Routines in MRST core are generally well documented in a format that follows the MATLAB standard. In addition, MRST supplies several worked tutorials that highlight functionality that we expect will be needed by most users; the tutorials are distributed as part of the MRST release and a subset of the tutorials are also available on the MRST webpage.

The add-on modules contain routines and functionality that extend, complement, and override existing MRST features, typically in the form of specialized or more advanced solvers and workflow tools like upscaling, grid coarsening, etc. In addition, there are modules that contain support for special input formats, Octave support, C-acceleration of selected routines in MRST core, etc. Some of these modules are robust, well-documented, and contain features that will likely not change in future releases, and could in principle had been included in MRST core if we had not decided to keep the core as small as possible. Examples of such modules are:

- consistent discretizations on general polyhedral grids: mimetic and MPFA-O methods;
- a small module for downloading and setting up flow models based on the SPE10 data set [19], which are commonly used benchmarks encountered in a multitude of papers;
- upscaling, including methods for flow-based single-phase upscaling as well as steady-state methods for two-phase upscaling;
- extended data structures for representing coarse grids as well as routines for partitioning logically Cartesian grids;
- agglomeration based grid coarsening: methods for defining coarse grids that adapt to geological features, flow patterns, etc;
- multiscale mixed finite-element methods for incompressible flow on stratigraphic and unstructured grids;
- multiscale finite-volume methods for incompressible flow on structured grids;
- the simplified deck reader, which contains support for processing simulation decks in the ECLIPSE format, including input reading, conversion to SI units, and construction MRST objects for grids, fluids, rock properties, and wells;
- C-acceleration of grid processing and basic solvers for incompressible flow.

Other modules and workflow tools, on the other hand, are constantly changing to support ongoing research:

- fully-implicit solvers based on automatic differentiation: rapid prototyping of flow models of industry-standard complexity;
- gui-based tools for interactive visualization of geological models and simulation results;
- flow diagnostics: simple, controlled numerical experiments that are run to probe a reservoir model and establish connections and basic volume estimates to compare, rank, and cluster models, or used as simplified flow proxies
- a numerical CO_2 that offers a chain of simulation tools of increasing complexity: geometrical methods for identifying structural traps, percolation type methods for identifying potential spill paths, and vertical-equilibrium methods that can efficiently simulate structural, residual, and solubility trapping in a thousand-year perspective;
- multiscale finite-volume methods for simulation of 'full physics' on stratigraphic and unstructured grids;
- production optimization based on adjoint methods.

All these modules are publicly available from the MRST webpage. In addition, there are several third-party modules that are available courtesy of their developers, as well as in-house prototype modules and workflow examples that are available upon request:

- two-point and multipoint solvers for discrete fracture-matrix systems, including multiscale methods, developed by researchers at the University of Bergen;
- an ensemble Kalman filter module developed by researchers at TNO, including EnKF and EnRML schemes, localization, inflation, asynchronous data, production and seismic data, updating of conventional and structural parameters;
- polymer flooding based on a Todd–Longstaff model with adsorption and dead pore space, permeability reduction, shear thinning, near-well (radial) and standard grids;
- geochemistry with conventional and structural parameters and without chemical equilibrium and coupling with fluid flow.

Discussing all these modules in detail is beyond the scope of the book. Instead, we encourge the interested reader to download and explore the modules on his/her own. Finally, the MRST webpage features several user-supplied cases and code examples that are not discussed in this book.

1.5 About examples and standard datasets

The examples play an important role in this book. Most examples have been equipped with codes the reader can rerun to reproduce the results discussed in the example. Likewise, many of the figures include a small box with the MATLAB and MRST commands necessary to create the plots. To the extent possible, we have tried to make the examples self-contained, but in some case we have for brevity omitted details that either have been discussed elsewhere or should be part of the reader's MATLAB basic repertoire.

MRST provides several routines that can be used to create examples in terms of grids and petrophysical data. The grid-factory routines are mostly deterministic and should enable the reader to create the exact same grids that are discussed in the book. The routines for generating petrophysical data, on the other hand, rely on random numbers, which means that the reader can only expect to reproduce plots and numbers that are qualitatively similar whenever these are used.

In addition, the book will use a few standard datasets that all can be downloaded freely from the internet. Herein, we use the convention that these datasets are stored in sub-directories of the following standard MRST path:

```
fullfile(ROOTDIR, 'examples', 'data')
```

We recommend that the reader adheres to this convention. In the following, we will briefly introduce the datasets and describe how to obtain them.

The SPE10 dataset

Model 2 of the 10th SPE Comparative Solution Project [19] was originally posed as a benchmark for upscaling methods. The 3-D geological model con-

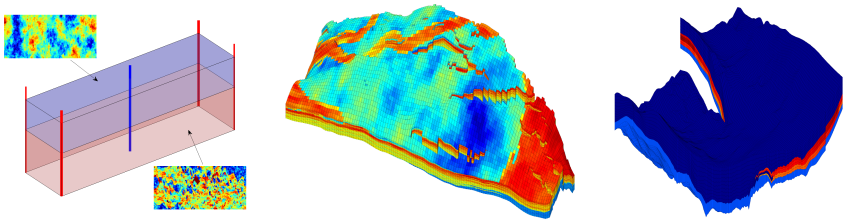


Fig. 1.2. Volumetric grid for three standard data sets used in the book: SPE10, SAIGUP, and Johansen.

sists of $60 \times 220 \times 85$ grid cells, each of size 20ft $\times$ 10ft $\times$ 2ft. The model is a geostatistical realization from the Jurassic Upper Brent formations, in which one can find the giant North Sea fields of Statfjord, Gullfaks, Oseberg, and Snorre. In this specific model, the top 70 ft (35 layers) represent the shallow-marine Tarbert formation and the lower 100 ft (50 layers) the fluvial Ness formation. The data can be obtained from the SPE website

http://www.spe.org/web/csp/

To simplify the reader's life, we supply a script `make_spe10_data` that checks if the SPE10 dataset exists on disk and if not, downloads and reorganizes the dataset and stores the result in the file `spe10_rock.mat`

SAIGUP dataset

As our primary example of a realistic petroleum reservoir, we will use a model from the SAIGUP study [40], whose purpose was to conduct a sensitivity analysis of the impact of geological uncertainties on production forecasting in clastic hydrocarbon reservoirs. As part of this study, a broad suite of geostatistical realizations and structural models were generated to represent a wide span of shallow-marine sedimentological reservoirs. All models are synthetic, but contain most of the type of complexities seen in real-life reservoirs. One particular out of the many SAIGUP realizations can be downloaded from the MRST website

http://www.sintef.no/Projectweb/MRST

The specific realization comes in the form of a GZip-compressed TAR file (`SAIGUP.tar.gz`) that contains the structural model as well as petrophysical parameters, represented using the corner-point grid format, which is an industry-standard in reservoir modelling.

The Johansen dataset

The Johansen formation is located in the deeper part of the Sognefjord delta, 40–90 km offshore Mongstad at the west coast of Norway. A gas-power plant

with carbon capture and storage is planned at Mongstad and the water-bearing Johansen formation is a possible candidate for storing the captured CO_2. The Johansen formation is part of the Dunlin group, and is interpreted as a large sandstone delta 2200–3100 meters below sea level that is limited above by the Dunlin shale and below by the Amundsen shale. The average thickness of the formation is roughly 100 m and the lateral extensions are up to 100 km in the north-south direction and 60 km in the east-west direction. With average porosities of approximately 25 percent, this implies that the theoretical storage capacity of the Johansen formation is more than one gigatons of CO_2 [28]. The Troll field, one of the largest gas field in the North Sea, is located some 500 meters above the north-western parts of the Johansen formation. Through a collaboration between the Norwegian Petroleum Directorate and researchers in the MatMoRA project, a set of geological models have been created and made available from the MatMoRA webpage:

http://www.sintef.no/Projectweb/MatMorA/Downloads/Johansen/

Altogether, there are five models: one full-field model ($149 \times 189 \times 16$ grid), three homogeneous sector models ($100 \times 100 \times n$ for $n = 11, 16, 21$), and one heterogeneous sector model ($100 \times 100 \times 11$).

Part I

Geological Models and Grids

2

Modelling Reservoir Rocks

Aquifers and natural petroleum reservoirs consist of a subsurface body of sedimentary rock having sufficient porosity and permeability to store and transmit fluids. In this chapter, we will given an overview of how such rocks are modelled to become part of a simulation model. We start by describing very briefly how sedimentary rocks are formed and then move on to describe how they are modelled. Finally, we discuss how rock properties are represented in MRST and show several examples of rock models with varying complexity, ranging from a homogeneous shoe-box rock body, via the widely used SPE 10 model, to two realistic models (one synthetic and one real-life).

2.1 Formation of a Sedimentary Reservoir

Sedimentary rocks are created by a process called sedimentation, in which mineral particles are broken off from a solid material by weathering and erosion in a source area and transported e.g., by water, to a place where they settle and accumulate to form what is called a sediment. Sediments may also contain organic particles that originate from the remains of plants, living creatures, and small organisms living in water, and may be deposited by other geophysical processes like wind, mass movement, and glaciers.

Sedimentary rocks have a layered structure with different mixtures of rock types with varying grain size, mineral types, and clay content. Over millions of years, layers of sediments built up in lakes, rivers, sand deltas, lagoons, choral reefs, and other shallow-marine waters; a few centimetres every hundred years formed sedimentary beds (or strata) that may extend many kilometres in the lateral directions. A bed denotes the smallest unit of rock that is distinguishable from an adjacent rock layer unit above or below it, and can be seen as bands of different color or texture in hillsides, cliffs, river banks, road cuts, etc. Each band represents a specific sedimentary environment, or mode of deposition, and can be from a few centimeters to several meters thick, often varying in the lateral direction. A sedimentary rock is characterized by

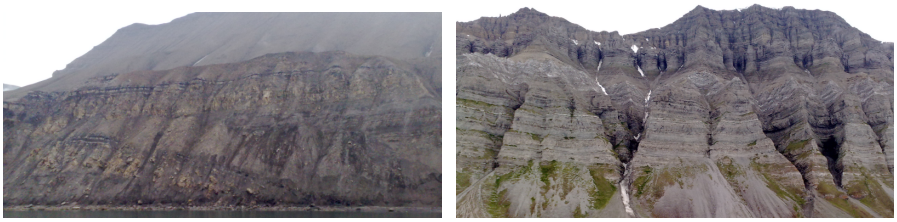


Fig. 2.1. Outcrops of sedimentary rocks from Svalbard, Norway. The length scale is a few hundred meters.

Fig. 2.2. Layered geological structures as seen in these pictures typically occur on both large and small scales in petroleum reservoirs. The right picture is courtesy of Silje Støren Berg, University of Bergen.

its bedding, i.e., sequence of beds and lamina (less pronounced layers). The bedding process is typically horizontal, but beds may also be deposited at a small angle, and parts of the beds may be weathered down or completely eroded way during deposition, allowing newer beds to form at an angle with older ones. Figure 2.1 shows two photos of sedimentary rock outcrops from Svalbard, which is one of the few places in Northern Europe where one can observe large-scale outcrops of sedimentary rocks. Figure 2.2 shows two more pictures of layered structures on meter and centimeter scales, respectively.

Each sedimentary environment has its own characteristic deposits and forms what is called a sedimentary facies, i.e., a body of rock with distinct characteristics. Different sedimentary environments usually exist alongside each other in a natural succession. Small stones, gravel and large sand particles are heavy and are deposited at the river bottom, whereas small sand particles are easily transported and are found at river banks and on the surrounding plains along with mud and clay. Following a rock layer of a given age, one will therefore see changes in the facies (rock type). Similarly, depositional environments change with time: shorelines move with changes in the sea level and land level or because of formation of river deltas, rivers change their course because of erosion or flooding, etc. Likewise, dramatic events like floods may

create abrupt changes. At a given position, the accumulated sequence of beds will therefore contain different facies.

As time passes by, more and more sediments accumulate and the stack of beds piles up to hundreds of meters. Simultaneously, severe geological activity takes place: Cracking of continental plates and volcanic activity changed what is to become our reservoir from being a relatively smooth, layered sedimentary basin into a complex structure where previously continuous layers were cut, shifted, or twisted in various directions, introducing fractures and faults. Fractures are cracks or breakage in the rock, across which there has been no movement; faults are fractures across which the layers in the rock have been displaced.

Over time, the depositional rock bodies got buried deeper and deeper, and the pressure and temperature increased because of the overburden. The deposits not only consisted of sand grains, mud, and small rock particles but also contained remains of plankton, algae, and other organisms living in the water that had died and fallen down to the bottom. As the organic material was compressed and 'cooked', it eventually turned into crude oil and natural gas. The lightest hydrocarbons (methane, ethane, etc.) usually escaped quickly, whilst the heavier oils moved slowly towards the surface. At certain sites, the migrating hydrocarbons were trapped in structural or stratigraphic traps. Structural traps (domes, folds, and anticlines) are created as geological activity deforms the layers containing hydrocarbon, near salt domes created by buried salt deposits that rise unevenly, or by sealing faults forming a closure. Stratigraphic traps form because of changes in facies (e.g., in clay content) within the bed itself or when the bed is sealed by an impermeable bed. These quantities of trapped hydrocarbons form todays oil and gas reservoirs. In the North Sea (which the authors are most familiar with), reservoirs are typically found 1 000–3 000 meters below the sea bed.

2.2 Multiscale Modelling of Permeable Rocks

All sedimentary rocks consist of a solid matrix with an interconnected void. The void pore space allows the rocks to store and transmit fluids. The ability to store fluids is determined by the volume fraction of pores (the rock porosity), and the ability to transmit fluids (the rock permeability) is given by the interconnection of the pores.

Rock formations found in natural petroleum reservoirs are typically heterogeneous at all length scales, from the micrometre scale of pore channels between the solid particles making up the rock to the kilometre scale of a full reservoir formation. On the scale of individual grains, there can be large variation in grain sizes, giving a broad distribution of void volumes and interconnections. Moving up a scale, laminae may exhibit large contrasts on the mm-cm scale in the ability to store and transmit fluids because of alternating layers of coarse and fine-grained material. Laminae are stacked to form beds,

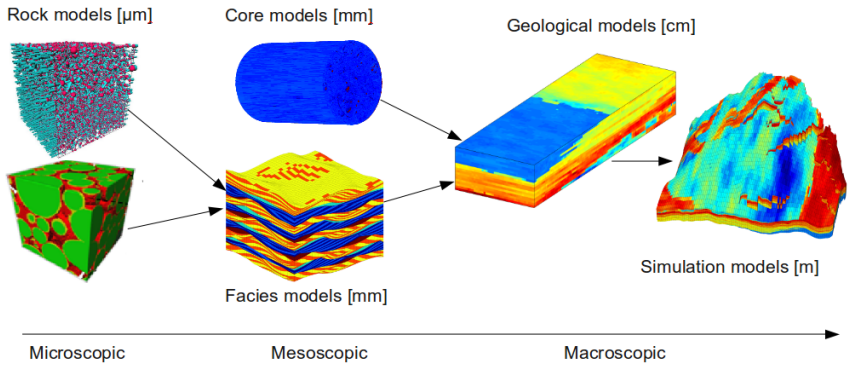


Fig. 2.3. Illustration of the hierarchy of flow models used in subsurface modeling. The length scales are the vertical sizes of typical elements in the models.

which are the smallest stratigraphic units. The thickness of beds varies from millimetres to tens of meters, and different beds are separated by thin layers with significantly lower permeability. Beds are, in turn, grouped and stacked into parasequences or sequences (parallel layers that have undergone similar geologic history). Parasequences represent the deposition of marine sediments, during periods of high sea level, and tend to be somewhere in the range from 1–100 meters thick and have a horizontal extent of several kilometres.

The trends and heterogeneity of parasequences depend on the depositional environment. For instance, whereas shallow-marine deposits may lead to rather smoothly varying permeability distributions with correlation lengths in the order 10–100 meters, fluvial reservoirs may contain intertwined patterns of sand bodies on a background with high clay content, see Figure 2.8. The reservoir geology can also consist of other structures like for instance shale layers (impermeable clays), which are the most abundant sedimentary rocks. Fractures and faults, on the other hand, are created by stresses in the rock and may extend from a few centimeters to tens or hundreds of meters. Faults may have a significantly higher or lower ability to transmit fluids than the surrounding rocks, depending upon whether the void space has been filled with clay material.

All these different length scales can have a profound impact on fluid flow. However, it is generally not possible to account for all pertinent scales that impact the flow in a single model. Instead, one has to create a hierarchy of models for studying phenomena occurring at reduced spans of scales. This is illustrated in Figure 2.3. Microscopic models represent the void spaces between individual grains and are used to provide porosity, permeability, electrical and elastic properties of rocks from core samples and drill cuttings. Mesoscopic models are used to upscale these basic rock properties from the mm/cm-scale of internal laminations, through the lithofacies scale ($\sim$ 50 cm), to the macro-

scopic facies association scale ($\sim$ 100 m) of geological models. In this book, we will primarily focus on another scale, simulation models, which represent the last scale in the model hierarchy. Simulation models are obtained by upscaling geological models and are either introduced out of necessity because geological models contain more details than a flow simulator can cope with, or out of convenience to provide faster calculation of flow responses.

2.2.1 Macroscopic Models

Geological models are built using a combination of stratigraphy (the study of rock layers and layering), sedimentology (study of sedimentary rocks), and interpretation of measured data. Unfortunately, building a geological model for a reservoir is like finishing a puzzle where most of the pieces are missing. Ideally, all available information about the reservoir is utilized, but the amount of data available is limited due to costs of acquiring them. Seismic surveys give a sort of X–ray image of the reservoir, but they are both expensive and time consuming, and can only give limited resolution (you cannot expect to see structures thinner than ten meters from seismic data). Wells give invaluable information, but the results are restricted to the vicinity of the well. While seismic has (at best) a resolution of ten meters, information on a finer scale are available from well-logs. Well-logs are basically data from various measuring tools lowered into the well to gather information, e.g., radiating the reservoir and measuring the response. Even well-logs give quite limited resolution, rarely down to centimetre scale. Detailed information is available from cores taken from wells, where resolution is only limited by the apparatus at hand. The industry uses X-ray, CT-scan as well as electron microscopes to gather high resolution information from the cores. However, information from cores and well-logs are from the well or near the well, and extrapolating this information to the rest of the reservoir is subject to great uncertainty. Moreover, due to costs, the amount of data acquisitions made is limited. You cannot expect well-logs and cores to be taken from every well. All these techniques give separately small contributions that can help build a geological model. However, in the end we still have very limited information available considering that a petroleum reservoir can have complex geological features that span across all types of length scales from a few millimetres to several kilometres.

In summary, the process of making a geological model is generally strongly under-determined. It is therefore customary to use *geostatistics* to estimate the subsurface characteristics between the wells. Using geostatistical techniques one builds petrophysical realizations in the form of grid models that both honor measured data and satisfy petrophysical trends and heterogeneity. Since trends and heterogeneity in petrophysical properties depend strongly on the structure of sedimentary deposits, high-resolution petrophysical realizations are not built directly. Instead, one starts by building a facies model. A facies is the characteristics of a rock unit that reflects its origin and separates

it from surrounding rock units. By supplying knowledge of the depositional environment (fluvial, shallow marine, deep marine, etc) and conditioning to observed data, one can determine the geometry of the facies and how they are mixed. In the second step, the facies are populated with petrophysical data and stochastic simulation techniques are used to simulate multiple realizations of the geological model in terms of high-resolution grid models for petrophysical properties. Each grid model has a plausible heterogeneity and can contain from a hundred thousand to a hundred million cells. The collection of all realizations gives a measure of the uncertainty involved in the modelling. Hence, if the sample of realizations (and the upscaling procedure that converts the geological models into simulation models) is unbiased, then it is possible to supply predicted production characteristics, such as the cumulative oil production, obtained from simulation studies with a measure of uncertainty.

This cursory overview of different models is all that is needed for what follows in the next few chapters, and the reader can therefore skip to Section 2.3 which discusses macroscopic modelling of reservoir rocks. The remains of this section will discuss microscopic and mesoscopic modelling in some more detail. First, however, we will briefly discuss the concept of representative elementary volumes, which underlies the continuum models used to describe subsurface flow and transport.

2.2.2 Representative Elementary Volumes

Choosing appropriate modelling scales is often done by intuition and experience, and it is hard to give very general guidelines. An important concept in choosing model scales is the notion of representative elementary volumes (REVs), which is the smallest volume over which a measurement can be made and be representative of the whole. This concept is based on the idea that petrophysical flow properties are constant on some intervals of scale, see Figure 2.4. Representative elementary volumes, if they exist, mark transitions between scales of heterogeneity, and present natural length scales for modelling.

To identify a range of length scales where REVs exist, e.g., for porosity, we move along the length-scale axis from the micrometer-scale of pores toward the kilometre-scale of the reservoir. At the pore scale, the porosity is a rapidly oscillating function equal to zero (in solid rock) or one (in the pores). Hence, obviously no REVs can exist at this scale. At the next characteristic length scale, the core scale level, we find laminae deposits. Because the laminae consist of alternating layers of coarse and fine grained material, we cannot expect to find a common porosity value for the different rock structures. Moving further along the length-scale axis, we may find long thin layers, perhaps extending throughout the entire horizontal length of the reservoirs. Each of these individual layers may be nearly homogeneous because they are created

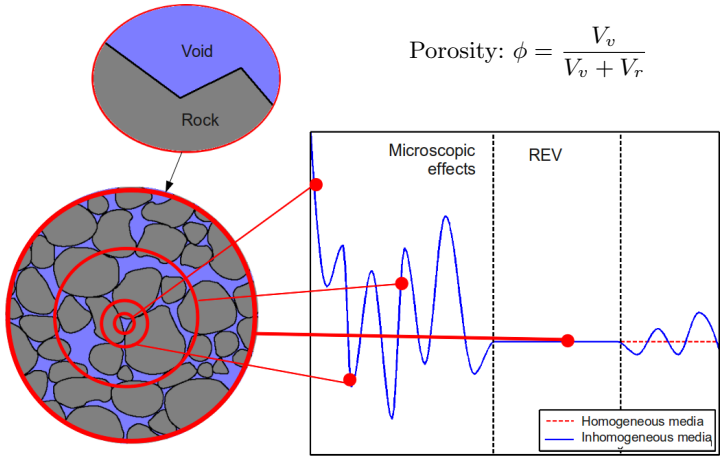


Fig. 2.4. The concept of a representative elementary volume (REV), here illustrated for porosity.

by the same geological process, and probably contain approximately the same rock types. Hence, at this scale it sounds reasonable to speak of an REV. If we move to the high end of the length-scale axis, we start to group more and more layers into families with different sedimentary structures, and REVs for porosity will probably not exist.

The previous discussion gives some grounds to claim that reservoir rock structures contain scales where REVs may exist. From a general point of view, however, the existence of REVs in porous media is highly disputable. A faulted reservoir, for instance, can have faults distributed continuously both in length and aperture throughout the reservoir, and will typically have no REVs. Moreover, no two reservoirs are identical, so it is difficult to capitalize from previous experience. Indeed, porous formations in reservoirs may vary greatly, also in terms of scales. Nevertheless, the concept of REVs can serve as a guideline when deciding what scales to model.

2.2.3 Microscopic Models: The Pore Scale

Pore-scale model, as illustrated to the left in Figure 2.3, may be about the size of a sugar cube and are based on measurements from core plugs obtained from well trajectories during drilling. These rock samples are necessarily confined (in dimension) by the radius of the well, although they lengthwise are only confined by the length of the well. Three such rock samples are shown in Figure 2.5. The main metods for obtaining pore-scale models from a rock sample is by studying thin slices using an electron microscope with micromete resolution or by CT-scans. In the following, we will give a simplified overview of flow modelling on this scale.

Fig. 2.5. Three core plugs with diameter of one and a half inches, and a height of five centimetres.

At the pore scale, the porous medium is either represented by a volumetric grid or by a graph (see e.g., [48]). A graph is a pair (V, E), where V is a set whose elements are called vertices (or nodes), and E is a subset of $V \times V$ whose elements are called edges. The vertices are taken to represent pores, and the edges represent pore-throats (i.e., connections between pores).

The flow process, in which one fluid invades the void space filled by another fluid, is generally described as an invasion–percolation process. This process is mainly dominated by capillary forces, although gravitational forces can still be important. In the invasion, a fluid phase can invade a pore only if a neighboring pore is already invaded. For each pore, there is an entry pressure, i.e., the threshold pressure needed for the invading phase to enter the pore, that depends on the size and shape of pores, the size of pore throats, as well as other rock properties. The invading phase will first invade the neighboring pore that has the lowest threshold pressure. This gives a way of updating the set of pores that are neighbors to invaded ones. Repeating the process establishes a recursive algorithm to determine the flow pattern of the invading phase. In the invasion process, we are interested in whether a phase has a path through the model, i.e., percolates, or not, and the time variable is often not modelled at all. For pore networks, this is misleading because we are also interested in modelling the flow after the first path through the model has been established. After a pore has been invaded, the saturations in the pore will vary with pressures and saturations in the neighboring pores (as well as in the pore itself). New pores may also be invaded after the first path is formed, so that we may get several paths through the model through which the invading phase can flow. Once the invading phase percolates (i.e., has a path through the model), one can start estimating flow properties. As the simulation progresses, the saturation of the invading phase will increase, which

can be used to estimate flow properties at different saturation compositions in the model.

In reality, the process is more complicated that explained above because of wettability. When two immiscible fluids (such as oil and water) contact a solid surface (such as the rock), one of them tends to spread on the surface more than the other. The fluid in a porous medium that preferentially contacts the rock is called the wetting fluid. Note that wettability conditions are usually changing throughout a reservoir. The flow process where the invading fluid is non-wetting is called drainage and is typically modelled with invasion–percolation. The flow process where the wetting fluid displaces the non-wetting fluid is called imbibition, and is more complex, involving effects termed film flow and snap-off.

Another approach to multiphase modelling is through the use of the lattice Boltzmann method that represents the fluids as a set of particles that propagate and collide according to a set of rules defined for interactions between particles of the same fluid phase, between particles of different fluid phases, and between the fluids and the walls of the void space. A further presentation of pore-scale modelling is beyond the scope here, but the interested reader is encouraged to consult, e.g., [48] and references therein.

From an analytical point of view, pore-scale modelling is very important as it represents flow at the fundamental scale (or more loosely, where the flow really takes place), and hence provides the proper framework for understanding the fundamentals of porous media flow. From a practical point of view, pore-scale modelling has a huge potential. Modelling flow at all other scales can be seen as averaging of flow at the pore scale, and properties describing the flow at larger scales are usually a mixture of pore-scale properties. At larger scales, the complexity of flow modelling is often overwhelming, with large uncertainties in determining flow parameters. Hence being able to single out and estimate the various factors determining flow parameters is invaluable, and pore-scale models can be instrumental in this respect. However, to extrapolate properties from the pore scale to an entire reservoir is very challenging, even if the entire pore space of the reservoir was known (of course, in real life you will not be anywhere close to knowing the entire pore space of a reservoir).

2.2.4 Mesoscopic Models

Models based on flow experiments on core plugs is by far the most common mesoscopic models. The fundamental equations describing flow are continuity of fluid phases and Darcy's law, which basically states that flow rate is proportional to pressure drop. The purpose of core-plug experiments is to determine capillary pressure curves and the proportionality constant in Darcy's law that measures the ability to transmit fluids, see (1.1) in Section 1.3. To this end, the sides of the core are insulated and flow is driven through the

core. By measuring the flow rate versus pressure drop, one can estimate the proportionality constant for both single-phase or multi-phase flows.

In conventional reservoir modelling, the effective properties from core-scale flow experiments are extrapolated to the macroscopic geological model, or directly to the simulation model. Cores should therefore ideally be representative for the heterogeneous structures that one may find in a typical grid block in the geological model. However, flow experiments are usually performed on relatively homogeneous cores that rarely exceed one meter in length. Cores can therefore seldom be classified as representative elementary volumes. For instance, cores may contain a shale barrier that blocks flow inside the core, but does not extend much outside the well-bore region, and the core was slightly wider, there would be a passage past the shale barrier. Flow at the core scale is also more influenced by capillary forces than flow on a reservoir scale.

As a supplement to core-flooding experiments, it has in recent years become popular to build 3D grid models to represent small-scale geological details like the bedding structure and lithology (composition and texture). One example of such a model is shown in Figure 2.3. Effective flow properties for the 3D model can now be estimated in the same way as for core plugs by replacing the flow experiment by flow simulations using rock properties that are e.g., based on the input from microscopic models. This way, one can incorporate fine-scale geological details from lamina into the macroscopic reservoir models.

This discussion shows that the problem of extrapolating information from cores to build a geological model is largely under-determined. Supplementary pieces of information are also needed, and the process of gathering geological data from other sources is described next.

2.3 Modelling of Rock Properties

How to describe the flow through a porous rock structure is largely a question of the scale of interest, as we saw in the previous section. The size of the rock bodies forming a typical petroleum reservoir will be from ten to hundred meters in the vertical direction and several hundred meters or a few kilometers in the lateral direction. On this modelling scale, it is clearly impossible to describe the storage and transport in individual pores and pore channels as discussed in Section 2.2.3 or the through individual lamina as in Section 2.2.4. To obtain a description of the reservoir geology, one builds models that attempt to reproduce the true geological heterogeneity in the reservoir rock at the macroscopic scale by introducing macroscopic petrophysical properties that are based on a continuum hypothesis and volume averaging over a sufficiently large representative elementary volume (REV), as discussed in Section 2.2.2. These petrophysical properties are engineering quantities that are used as input to flow simulators and are not geological or geophysical properties of the underlying media.

A geological model is a conceptual, three-dimensional representation of a reservoir, whose main purpose is therefore to provide the distribution of petrophysical parameters, besides giving location and geometry of the reservoir. The rock body itself is modelled in terms of a volumetric grid, in which the layered structure of sedimentary beds and the geometry of faults and large-scale fractures in the reservoir are represented by the geometry and topology of the grid cells. The size of a cell in a typical geological grid-model is in the range of 0.1–1 meters in the vertical direction and 10–50 meters in the horizontal direction. The petrophysical properties of the rock are represented as constant values inside each grid cell (porosity and permeability) or as values attached to cell faces (fault multipliers, fracture apertures, etc). In the following, we will describe the main rock properties in more detail. More details about the grid modelling will follow in Chapter 3.

2.3.1 Porosity

The porosity ϕ of a porous medium is defined as the fraction of the bulk volume that is occupied by void space, which means that $0 \leq \phi < 1$. Likewise, $1 - \phi$ is the fraction occupied by solid material (rock matrix). The void space generally consists of two parts, the interconnected pore space that is available to fluid flow, and disconnected pores (dead-ends) that is unavailable to flow. Only the first part is interesting for flow simulation, and it is therefore common to introduce the so-called "effective porosity" that measures the fraction of connected void space to bulk volume.

For a completely rigid medium, porosity is a static, dimensionless quantity that can be measured in the absence of flow. Porosity is mainly determined by the pore and grain-size distribution. Rocks with nonuniform grain size typically have smaller porosity than rocks with a uniform grain size, because smaller grains tend to fill pores formed by larger grains. Similarly, for a bed of solid spheres of uniform diameter, the porosity depends on the packing, varying between 0.2595 for a rhombohedral packing to 0.4764 for cubic packing. For most naturally-occuring rocks, ϕ is in the range 0.1–0.4, although values outside this range may be observed on occasion. Sandstone porosity is usually determined by the sedimentological process by which the rock was deposited, whereas for carbonate porosity is mainly a result of changes taking place after deposition.

For non-rigid rocks, the porosity is usually modelled as a pressure-dependent parameter. That is, one says that the rock is *compressible*, having a rock compressibility defined by:

$$c_r = \frac{1}{\phi}\frac{d\phi}{dp} = \frac{d\ln(\phi)}{dp}, \tag{2.1}$$

where p is the overall reservoir pressure. Compressibility can be significant in some cases, e.g., as evidenced by the subsidence observed in the Ekofisk area in the North Sea. For a rock with constant compressibility, (2.1) can be integrated to give

$$\phi(p) = \phi_0 e^{c_r(p-p_0)}, \tag{2.2}$$

and for simplified models, it is common to use a linearization so that:

$$\phi = \phi_0 \big[1 + c_r(p - p_0)\big]. \tag{2.3}$$

Because the dimension of the pores is very small compared to any interesting scale for reservoir simulation, one normally assumes that porosity is a piecewise continuous spatial function. However, ongoing research aims to understand better the relation between flow models on pore scale and on reservoir scale.

2.3.2 Permeability

The *permeability* is the basic flow property of a porous medium and measures its ability to transmit a single fluid when the void space is completely filled with this fluid. This means that permeability, unlike porosity, is a parameter that cannot be defined apart from fluid flow. The precise definition of the (absolute, specific, or intrinsic) permeability K is as the proportionality factor between the flow rate and an applied pressure or potential gradient $\nabla\Phi$,

$$\vec{u} = -\frac{K}{\mu}\nabla\Phi. \tag{2.4}$$

This relationship is called Darcy's law after the french hydrologist Henry Darcy, who first observed it in 1856 while studying flow of water through beds of sand [21]. In (2.4), μ is the fluid viscosity and $\vec{u}$ is the superficial velocity, i.e., the flow rate divided by the cross-sectional area perpendicular to the flow, which should not be confused with the interstitial velocity $\vec{v} = \phi^{-1}\vec{u}$, i.e., the rate at which an actual fluid particle moves through the medium. We will come back to a more detailed discussion of Darcy's law in Section 5.2.

The SI-unit for permeability is m^2, which reflects the fact that permeability is determined by the geometry of the medium. However, it is more common to use the unit 'darcy' (D). The precise definition of 1D ($\approx 0.987 \cdot 10^{-12}\,\mathrm{m}^2$) involves transmission of a 1cp fluid through a homogeneous rock at a speed of 1cm/s due to a pressure gradient of 1atm/cm. Translated to reservoir conditions, 1D is a relatively high permeability and it is therefore customary to specify permeabilities in milli-darcies (mD). Rock formations like sandstones tend to have many large or well-connected pores and therefore transmit fluids readily. They are therefore described as permeable. Other formations, like shales, may have smaller, fewer or less interconnected pores and are hence described as impermeable. Conventional reservoirs typically have permeabilities ranging from 0.1 mD to 20 D for liquid flow and down to 10 mD for gas flow. In recent years, however, there has been an increasing interest in unconventional resources, that is, gas and oil locked in extraordinarily impermeable and hard rocks, with permeability values ranging from 0.1 mD and down to 1

μD or lower. Compared with conventional resources, the potential volumes of tight gas, shale gas, shale oil are enormous, but cannot be easily produced at economic rates unless stimulated, e.g., using a pressurized fluid to fracture the rock (hydraulic fracturing). In this book, our main focus will be on simulation of conventional resources.

In general, the permeability is a tensor, which means that the permeability in the different directions depends on the permeability in the other directions. The tensor is represented by a matrix in which the diagonal terms represent direct flow, i.e., flow in one direction caused by a pressure drop in the same direction. The off-diagonal terms represent cross-flow, i.e., flow caused by pressure drop in directions perpendicular to the flow. A full tensor is needed to model local flow in directions at an angle to the coordinate axes. For example, in a layered system the dominant direction of flow will generally be along the layers but if the layers form an angle to the coordinate axes, then a pressure drop in one coordinate direction will produce flow at an angle to this direction. This type of flow can be modelled correctly only with a permeability tensor with nonzero off-diagonal terms. However, by a change of basis, K may sometimes be diagonalized, and because of the reservoir structure, horizontal and vertical permeability suffices for several models. We say that the medium is isotropic (as opposed to anisotropic) if K can be represented as a scalar function, e.g., if the horizontal permeability is equal to the vertical permeability.

The permeability is obviously a function of porosity. Assuming a laminar flow (low Reynolds numbers) in a set of capillary tubes, one can derive the Carman–Kozeny relation,

$$K = \frac{1}{8\tau A_v^2} \frac{\phi^3}{(1-\phi)^2}, \tag{2.5}$$

which relates permeability to porosity ϕ, but also shows that the permeability depends on local rock texture described by tortuosity τ and specific surface area A_v. The tortuosity is defined as the squared ratio of the mean arc-chord length of flow paths, i.e., the ratio between the length of a flow path and the distance between its ends. The specific surface area is an intrinsic and characteristic property of any porous medium that measures the internal surface of the medium per unit volume. Clay minerals, for instance, have large specific surface areas and hence low permeability. The quantities τ and A_v can be calculated for simple geometries, e.g., for engineered beds of particles and fibers, but are seldom measured for reservoir rocks. Moreover, the relationship in (2.5) is highly idealized and only gives satisfactory results for media that consist of grains that are approximately spherical and have a narrow size distribution. For consolidated media and cases where rock particles are far from spherical and have a broad size-distribution, the simple Carman–Kozeny equation does not apply. Instead, permeability is typically obtained through macroscopic flow measurements.

Permeability is generally heterogeneous in space because of different sorting of particles, degree of cementation (filling of clay), and transitions between different rock formations. Indeed, the permeability may vary rapidly over several orders of magnitude, local variations in the range 1 mD to 10 D are not unusual in a typical field. The heterogeneous structure of a porous rock formation is a result of the deposition and geological history and will therefore vary strongly from one formation to another, as we will see in a few of the examples in Section 2.4.

Production of fluids may also change the permeability. When temperature and pressure is changed, microfractures may open and significantly change the permeability. Furthermore, since the definition of permeability involves a certain fluid, different fluids will experience different permeability in the same rock sample. Such rock-fluid interactions are discussed in Chapter **??**.

2.3.3 Other parameters

Not all rocks in a reservoir zone are reservoir rocks. To account for the fact that some portion of a cell may consist of impermeable shale, it is common to introduce the so-called "net-to-gross" (N/G) property, which is a number in the range 0 to 1 that represents the fraction of reservoir rock in the cell. To get the effective porosity of a given cell, one must multiply the porosity and N/G value of the cell. (The N/G values also act as multipliers for lateral transmissibilities, which we will come back to later in the book). A zero value means that the corresponding cell only contains shale (either because the porosity, the N/G value, or both are zero), and such cells are by convention typically not included in the active model. Inactive cells can alternatively be specified using a dedicated field (called 'actnum' in industry-standard input formats).

Faults can either act as conduits for fluid flow in subsurface reservoirs or create flow barriers and introduce compartmentalization that severely affects fluid distribution and/or reduces recovery. On a reservoir scale, faults are generally volumetric objects that can be described in terms of displacement and petrophysical alteration of the surrounding host rock. However, lack of geological resolution in simulation models means that fault zones are commonly modelled as surfaces that explicitly approximate the faults' geometrical properties. To model the hydraulic properties of faults, it is common to introduce so-called multipliers that alter the ability to transmit fluid between two neighboring cells. Multipliers are also used to model other types of subscale features that affect communication between grid blocks, e.g., thin mud layers resulting from flooding even which may partially cover the sand bodies and reduce vertical communication. It is also common to (ab)use multipliers to increase or decrease the flow in certain parts of the model to calibrate the simulated reservoir responses to historic data (production curves from wells, etc). More details about multipliers will be given later in the book.

2.4 Rock Modelling in MRST

All flow and transport solvers in MRST assume that the rock parameters are represented as fields in a structure. Our naming convention is that this structure is called `rock`, but this is not a requirement. The fields for porosity and permeability, however, must be called `poro` and `perm`, respectively. The porosity field `rock.poro` is a vector with one value for each active cell in the corresponding grid model. The permeability field `rock.perm` can either contain a single column for an isotropic permeability, two or three columns for a diagonal permeability (in two and three spatial dimensions, respectively, or six columns for a symmetric, full tensor permeability. In the latter case, cell number i has the permeability tensor

$$\mathsf{K}_i = \begin{bmatrix} K_1(i) & K_2(i) \\ K_2(i) & K_3(i) \end{bmatrix}, \qquad \mathsf{K}_i = \begin{bmatrix} K_1(i) & K_2(i) & K_3(i) \\ K_2(i) & K_4(i) & K_5(i) \\ K_3(i) & K_5(i) & K_6(i) \end{bmatrix},$$

where $K_j(i)$ is the entry in column j and row i of `rock.perm`. Full-tensor, non-symmetric permeabilities are currently not supported in MRST. In addition to porosity and permeability, MRST supports a field called `ntg` that represents the net-to-gross ratio and consists of either a scalar or a single column with one value per active cell.

In the rest of the section, we present a few examples that demonstrate how to generate and specify permeability and porosity values. In addition, we will briefly discuss a few models with industry-standard complexity. Through the discussion, you will also be exposed to a lot of the visualization capabilities of MRST. Complete scripts necessary to reproduce the results and the figures presented can be found in various scripts in the `rock` subdirectory of the software module that accompanies the book.

2.4.1 Homogeneous Models

Homogeneous models are very simple to specify, as is illustrated by a simple example. We consider a square 10×10 grid model with a uniform porosity of 0.2 and isotropic permeability equal 200 mD:

```
G = cartGrid([10 10]);
rock.poro = repmat( 0.2, [G.cells.num,1]);
rock.perm = repmat( 200*milli*darcy, [G.cells.num,1]);
```

Because MRST works in SI units, it is important to convert from the field units 'darcy' to the SI unit 'meters^2'. Here, we did this by multiplying with `milli` and `darcy`, which are two functions that return the corresponding conversion factors. Alternatively, we could have used the conversion function `convertFrom`(200, `milli*darcy`). Homogeneous, anisotropic permeability can be specified in the same way:

```
rock.perm = repmat( [100 100 10].*milli*darcy, [G.cells.num,1]);
```

2.4.2 Random and Lognormal Models

Given the difficulty of measuring rock properties, it is common to use geostatistical methods to make realizations of porosity and permeability. MRST contains two *very* simplified methods for generating geostatistical realizations. For more realistic geostatistics, the reader should use GSLIB [24] or a commercial geomodelling software.

In our first example, we will generate the porosity ϕ as a Gaussian field. To get a crude approximation to the permeability-porosity relationship, we assume that our medium is made up of uniform spherical grains of diameter $d_p = 10\,\mu$m, for which the specific surface area is $A_v = 6/d_p$. Using the Carman–Kozeny relation (2.5), we can then calculate the isotropic permeability K from

$$K = \frac{1}{72\tau}\frac{\phi^3 d_p^2}{(1-\phi)^2},$$

where we further assume that $\tau = 0.81$. As a simple approximation to a Gaussian field, we generate a field of independent normally distributed variables and convolve it with a Gaussian kernel.

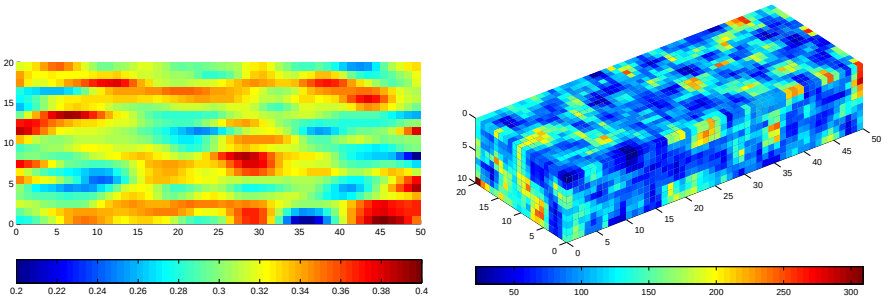
```
G = cartGrid([50 20]);
p = gaussianField(G.cartDims, [0.2 0.4], [11 3], 2.5);
K = p.^3.*(1e-5)^2./(0.81*72*(1-p).^2);
rock.poro = p(:);
rock.perm = K(:);
```

The resulting porosity field is shown in the left plot of Figure 2.6. The right plot shows the permeability obtained for a 3D realization generated in the same way.

In the second example, we use the same methodology as above to generate layered realizations, for which the permeability in each geological layer is independent of the other layers and lognormally distributed. Each layer can be represented by several grid cells in the vertical direction. Rather than using a simple Cartesian grid, we will generate a stratigraphic grid with wavy geological faces and a single fault. Such grids will be described in more detail in Chapter 3.

```
G = processGRDECL(simpleGrdecl([50 30 10], 0.12));
K = logNormLayers(G.cartDims, [100 400 50 350], ...
                  'indices', [1 2 5 7 11]);
```

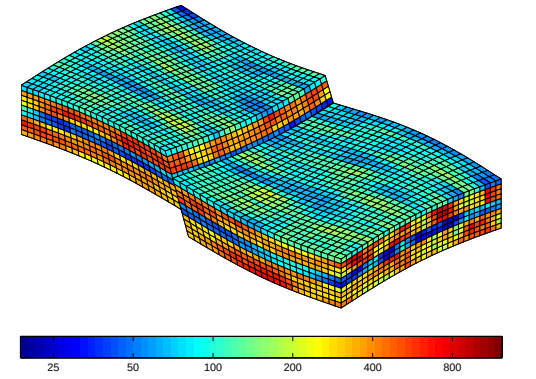
Here we have specified four geological layers with mean values of 100 mD, 400 mD, 50 mD, and 350 mD from top to bottom (stratigraphic grids are numbered from the top and downward). The layers are represented with one, three, two, and four grid cells, respectively, in the vertical direction. The resulting permeability is shown in Figure 2.7.



```
plotCellData (G, rock.poro );
colorbar ('horiz'); axis equal tight;
```

```
plotCellData (G, convertTo ( rock.perm , milli * darcy ));
colorbar ('horiz'); axis equal tight; view (3);
```

Fig. 2.6. The left plot shows a 50×20 porosity field generated as a Gaussian field with a larger filter size in x-direction than in the y-direction. The right plot shows the permeability field computed from the Carman–Kozeny relation for a similar $50 \times 20 \times 10$ porosity realization computed with filter size $[3, 3, 3]$.



```
plotCellData (G, log10 (K), 'EdgeColor','k'); view (45,30);
axis tight off, set(gca,'DataAspect',[0.5 1 1])
h=colorbar('horiz'); ticks=25*2.^[0:5];
set(h,'XTick',log10(ticks),'XTickLabel',ticks);
```

Fig. 2.7. A stratigraphic grid with a single fault and four geological layers, each with a lognormal permeability distribution.

2.4.3 10th SPE Comparative Solution Project: Model 2

The model was originally posed as a benchmark for upscaling method, but has later become very popular within the academic community as a benchmark for comparing different computational methods. The model is structurally simple but is highly heterogeneous, and, for this reason, some describe it as a 'simulator-killer'. On the other hand, the fact that the flow is dictated by the strong heterogeneity means that streamline methods will be particularly

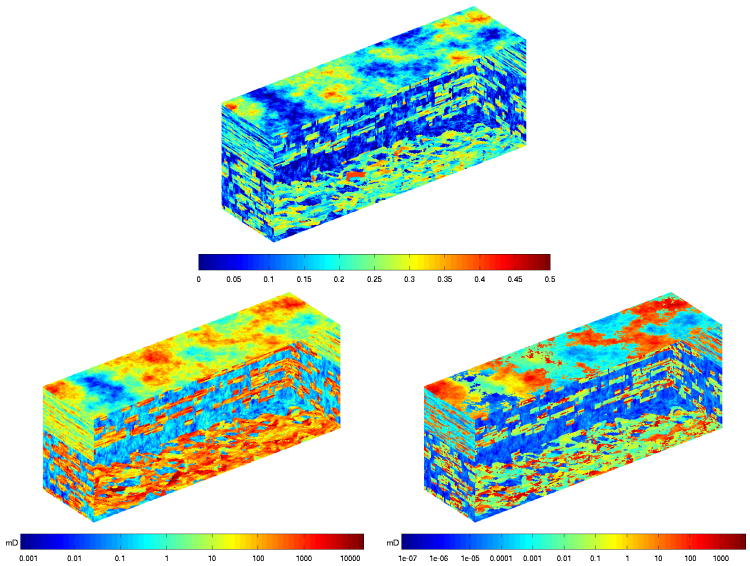


Fig. 2.8. Rock properties for the SPE 10 model. The upper plot shows the porosity, the lower left the horizontal permeability, and the lower right the vertical permeability. (The permeabilities are shown using a logarithmic color scale).

efficient for this model [1]. The SPE 10 data set is used in a large number of publications, and for this reason we have made a module `spe10` in MRST that downloads and provides simple access to this model. Because the geometry is a simple Cartesian grid, we can use standard MATLAB functionality to visualize the heterogeneity in the permeability and porosity (full details can be found in the script `rocks/showSPE10.m`)

```
% load SPE 10 data set
mrstModule add spe10;
rock = SPE10_rock(); p=rock.poro; K=rock.perm;

% show p
slice( reshape(p,60,220,85), [1  220],  60,  [1  85]);
shading flat, axis equal off, set(gca,'zdir','reverse'),  box on;
colorbar('horiz');

% show Kx
slice( reshape(log10(K(:,1)),60,220,85),  [1  220],  60,  [1  85]);
shading flat, axis equal off, set(gca,'zdir','reverse'),  box on;
h=colorbar('horiz');
set(h,'XTickLabel',10.^[get(h,'XTick')]);
set(h,'YTick',mean(get(h,'YLim')),'YTickLabel','mD');
```

Figure 2.8 shows porosity and permeability; the permeability tensor is diagonal with equal permeability in the two horizontal coordinate directions.

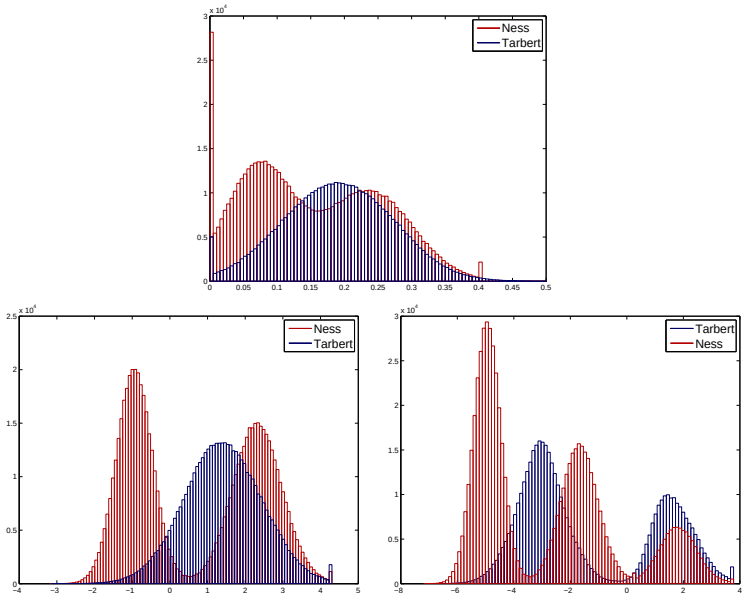


Fig. 2.9. Histogram of rock properties for the SPE 10 model: ϕ (upper plot), $\log \mathbf{K}_x$ (lower left), and $\log \mathbf{K}_z$ (lower right) The Tarbert formation is shown in blue and the Ness formation in red.

Both formations are characterized by large permeability variations, 8–12 orders of magnitude, but are qualitatively different. The Tarbert consists of sandstone, siltstone, and shales and comes from a tidally influenced, transgressive, shallow-marine deposit. The formation has good communication in the vertical and horizontal directions. The fluvial Ness formation has been deposited by rivers or running water in a delta-plain environment, leading to a spaghetti of well-sorted high-permeable sandstone channels with good communication (long correlation lengths) imposed on a low-permeable background of shales and coal, which gives low communication between different sand bodies. The porosity field has a large span of values and approximately 2.5% of the cells have zero porosity and should be considered as being inactive.

Figure 2.9 shows histograms of the porosity and the logarithm of the horizontal and vertical permeabilities. The nonzero porosity values and the horizontal permeability of the Tarbert formation appear to follow a normal and lognormal distribution, respectively. The vertical permeability follows a bimodal distribution. For the Ness formation, the nonzero porosities and the horizontal permeability follow bi-modal normal and lognormal distributions, respectively, as is to be expected for a fluvial formation. The vertical permeability is tri-modal.

2.4.4 The Johansen Formation

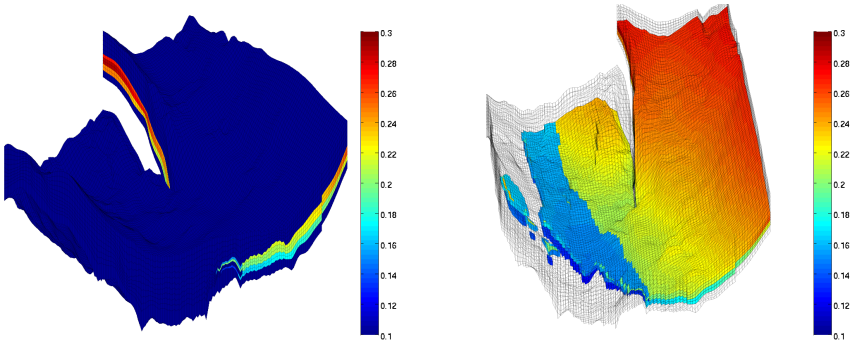
The Johansen formation is a candidate site for large-scale CO_2 storage offshore the south-west coast of Norway; with good sand quality and depth levels between 2200 and 3100 meters, the pressure regime should be ideal for CO_2 storage. Herein, we will consider the heterogeneous sector model given as a $100 \times 100 \times 11$ corner-point grid. All statements used to analyze the model are found in the script `rocks/showJohansenNPD5.m`.

The grid consists of hexahedral cells and is given on the industry-standard corner-point format, which will be discussed in details in Section 3.3.1. A more detailed discussion of how to input the grid will be given in the next section. The rock properties are given as plain ASCII files, with one entry per cell. In the model, the Johansen formation is represented by five grid layers, the low-permeable Dunlin shale above is represented by five layers, and the Amundsen shale below is represented as one layer. The Johansen formation consists of approximately 80% sandstone and 20% claystone, whereas the Amundsen formation consists of siltstones and shales, see [28, 27, 8] for more details.

We start by loading the data and visualizing the porosity, which is straightforward once we remember to use `G.cells.indexMap` to extract rock properties only for active cells in the model.

```
G = processGRDECL(readGRDECL('NPD5.grdecl'));
p = load('NPD5_Porosity.txt')'; p = p(G.cells.indexMap);
```

Figure 2.10 shows the porosity field of the model. The left plot shows the Dunlin shale, the Johansen sand, and the Amundsen shale, where the Johansen




```
args = {'EdgeAlpha', 0.1, 'EdgeColor', 'k'};
plotCellData(G, p, args{:}); view(-45,15),
axis tight off, zoom(1.15),
caxis([0.1 0.3]), colorbar;
```

```
plotGrid(G,'FaceColor','none', args{:});
plotCellData(G, p, find(p>0.1), args{:})
view(-15,40); axis tight off, zoom(1.15),
caxis([0.1 0.3]), colorbar;
```

Fig. 2.10. Porosity for the Johansen data set 'NPD5'. The left plot shows porosity for the whole model, whereas in the right plot we have masked the low-porosity cells in the Amundsen and Dunlin formations.

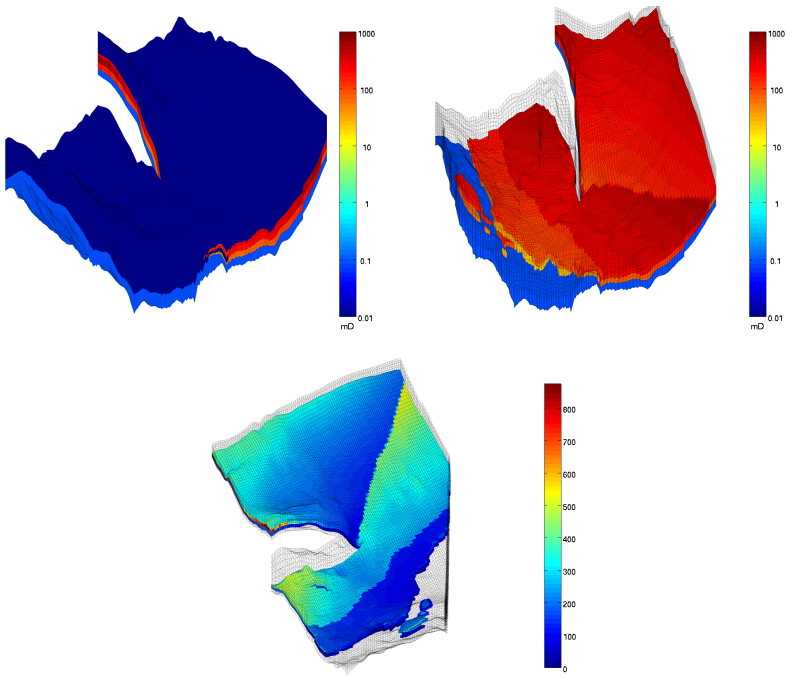


Fig. 2.11. Permeability for the Johansen data set 'NPD5'. The upper-left plot shows the permeability for the whole model, the upper-right plot shows the Johansen sand and the Amundsen shale, whereas the lower plot only shows the permeability of the Johansen sand.

sand is clearly distinguished as a wedge shape that is pinched out in the front part of the model and splits the shales laterally in two at the back. In the right plot, we only plot the good reservoir rocks distinguished as the part of the porosity field that has values larger than 0.1.

The permeability tensor is assumed to be diagonal with the vertical permeability equal one-tenth of the horizontal permeability. Hence, only the x-component $\mathbf{K}_x$ is given in the data file

```
K = load('NPD5_Permeability.txt')'; K=K(G.cells.indexMap);
```

Figure 2.11 shows three different plots of the permeability. The first plot shows the logarithm of whole permeability field. In the second plot, we have filtered out the Dunlin shale above Johansen but not the Amundsen shale below. The third plot shows the permeability in the Johansen formation using a linear color scale, which clearly shows the depth trend that was used to model the heterogeneity.

2.4.5 The SAIGUP Model

Most commercial simulators use a combination of an 'input language' and a set of data files to describe and set up a simulation model of a reservoir. However, although the principles for the input description has much in common, the detail syntax is obviously unique to each simulator. Herein, we will mainly focus on the ECLIPSE input format, which has emerged as an industry standard for describing static and dynamic properties of a reservoir system, from the reservoir rock, via production and injection wells and up to connected topside facilities. ECLIPSE input decks use keywords to signify and separate the different data elements that comprise a full model. These keywords define a detailed language that can be used to specify how the data elements should be put together, and modify each other, to form a full spatio-temporal model of a reservoir. In the most general form, an ECLIPSE input file consists of eight sets of keywords are organized into eight sections that must come in a prescribed order. However, some of the sections are optional and may not always be present. The order of the keywords within each section is arbitrary, except in the section that defines wells and gives operating schedule, etc. Altogether, the ECLIPSE format consists of thousands of keywords, and describing them all is far beyond the scope of this presentation.

In the following, we will instead briefly outline some of the most common keywords that are used in the GRID section that describes the reservoir geometry and petrophysical properties. The purpose is to provide you with a basic understanding of the required input for simulations of real-life reservoir models. Our focus is mainly on the ingredients of a model and not on the specific syntax. For brevity, we will therefore not go through all Matlab and MRST statements used to visualize the different data elements. All details necessary to reproduce the results can be found in the script `rocks/showSAIGUP.m`.

As an example of a realistic representation of a shallow-marine reservoir, we will use a simulation model taken from the SAIGUP study [40]. The SAIGUP models mainly focus on shoreface reservoirs in which the deposition of sediments is caused by variation in sea level, so that facies are forming belts in a systematic pattern (river deposits create curved facies belts, wave deposits create parallel belts, etc). Sediments are in general deposited when the sea level is increasing. No sediments are deposited during decreasing sea levels; instead, the receding sea may affect the appearing shoreline and cause the creation of a barrier.

Assuming that the archive file `SAIGUP.tar.gz` that contains the model realization has been downloaded as described in Section 1.5, we used MATLAB's `untar` function to extract the data set and place it in a standardized path relative to the root directory of MRST:

```
untar('SAIGUP.tar.gz', fullfile(ROOTDIR, 'examples', 'data', 'SAIGUP'))
```

This will create a new directory that contains seventeen data files that comprise the structural model, various petrophysical parameters, etc:

```
028_A11.EDITNNC      028.MULTX  028.PERMX  028.SATNUM       SAIGUP.GRDECL
028_A11.EDITNNC.001  028.MULTY  028.PERMY  SAIGUP_A1.ZCORN
028_A11.TRANX        028.MULTZ  028.PERMZ  SAIGUP.ACTNUM
028_A11.TRANY        028.NTG    028.PORO   SAIGUP.COORD
```

The main file is `SAIGUP.GRDECL`, which lists the sequence of keywords that specifies how the data elements found in the other files should be put together to make a complete model of the reservoir rock. The remaining files represent different keywords: the grid geometry is given in files `SAIGUP_A1.ZCORN` and `SAIGUP.COORD`, the porosity in `028.PORO`, the permeability tensor in the three `028.PERM*` files, net-to-gross properties in `028.NTG`, the active cells in `SAIGUP.ACTNUM`, transmissibility multipliers that modify the flow connections between different cells in the model are given in `028.MULT*`, etc. For now, we will rely entirely on MRST's routines for reading Eclipse input files; more details about corner-point grids and the Eclipse input format will follow later in the book, starting in Chapter 3.

The `SAIGUP.GRDECL` file contains seven of the eight possible sections that may comprise a full input deck. The `deckformat` module in MRST contains a comprehensive set of input routines that enable the user to read the most important keywords and options supported in these sections. Here, however, it is mainly the sections describing static reservoir properties that contain complete and useful information, and we will therefore use the much simpler function `readGRDECL` from MRST core to read and interprets the GRID section of the input deck:

```
grdecl = readGRDECL(fullfile(ROOTDIR, 'examples', ...
   'data', 'SAIGUP','SAIGUP.GRDECL'));
```

This statement parses the input file and stores the content of all keywords it recognizes in the structure `grdecl`:

```
grdecl =
    cartDims: [40 120 20]
       COORD: [29766x1 double]
       ZCORN: [768000x1 double]
      ACTNUM: [96000x1 int32]
       PERMX: [96000x1 double]
       PERMY: [96000x1 double]
       PERMZ: [96000x1 double]
       MULTX: [96000x1 double]
       MULTY: [96000x1 double]
       MULTZ: [96000x1 double]
        PORO: [96000x1 double]
         NTG: [96000x1 double]
      SATNUM: [96000x1 double]
```

The first four data fields describe the grid, and we will come back to these in Chapter 3.3.1. In the following, we will focus on the next eight data fields, which contain the petrophysical parameters. We will also briefly look at the last data field, which delineates the reservoir into different (user-defined) rock types that can used to associated different rock-fluid properties.

MRST uses the strict SI conventions in all of its internal calculations. The SAIGUP model, however, is provided using the Eclipse 'METRIC' conventions

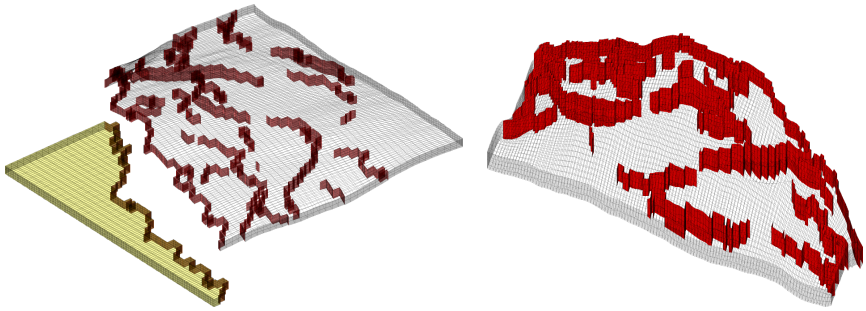


Fig. 2.12. The structural SAIGUP model. The left plot shows the full model with faults marked in red and inactive cells marked in yellow, whereas the right plot shows only the active parts of the model seen from the opposite direction.

(permeabilities in mD and so on). We use the functions `getUnitSystem` and `convertInputUnits` to assist in converting the input data to MRST's internal unit conventions.

```
usys   = getUnitSystem('METRIC');
grdecl = convertInputUnits(grdecl, usys);
```

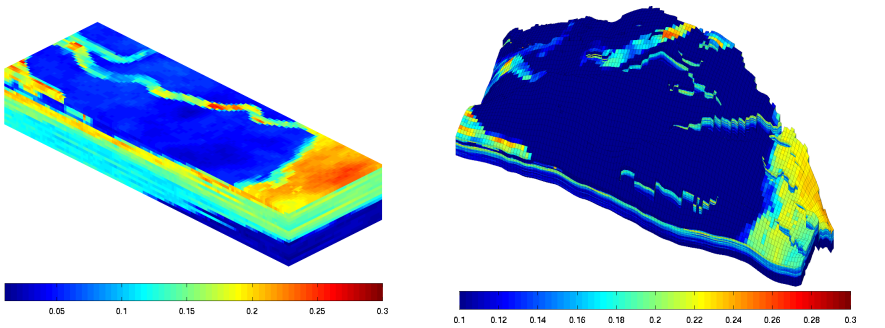
Having converted the units properly, we generate a space-filling grid and extract petrophysical properties

```
G = processGRDECL(grdecl);
G = computeGeometry(G);
rock   = grdecl2Rock(grdecl, G.cells.indexMap);
```

The first statement takes the description of the grid geometry and constructs an unstructured MRST grid represented with the data structure outlined in Section 3.4. The second statement computes a few geometric primitives like cell volumes, centroids, etc., as discussed on page 75. The third statement constructs a rock object containing porosity, permeability, and net-to-gross.

For completeness, we first show a bit more details of the structural model in Figure 2.12. The left plot shows the whole $40 \times 120 \times 20$ grid model[1], where we in particular should note the disconnected cells marked in yellow that are not part of the active model. The relatively large fault throw that disconnects the two parts is most likely a modelling artifact introduced to clearly distinguish the active and inactive parts of the model. A shoreface reservoir is bounded by faults and geological horizons, but faults also appear inside the reservoir as the right plot in Figure 2.12 shows. Faults and barriers will typically have a pronounced effect on the flow pattern, and having an accurate representation is important to produce reliable flow predictions.

[1] To not confuse the reader, we emphasize that only the active part of the model is read with the MRST statements given above. How to also include the inactive part, will be explained in more details in Chapter 3.




```
p = reshape( grdecl.PORO , G.cartDims );
slice(p, 1, 1, 1); view(-135,30), shading flat,
axis equal off,
set(gca,'ydir','reverse','zdir','reverse')
colorbar('horiz'); caxis([0.01 0.3]);
```

```
args = {'EdgeAlpha', 0.1, 'EdgeColor', 'k'};
plotCellData(G, rock.poro , args{:});
axis tight off; set(gca,'DataAspect',[1 1 0.1]);
view(-65,55); zoom(1.4); camdolly(0,-0.2,0)
colorbar('horiz'); caxis([0.1 0.3])
```

Fig. 2.13. Porosity for the SAIGUP model. The left plot shows porosity as generated by geostatistics in logical ijk space. The right plot shows the porosity mapped to the structural model shown in Figure 2.12.

The petrophysical parameters for the model were generated on a regular $40 \times 120 \times 20$ Cartesian grid, as illustrated in the left plot of Figure 2.13, and then mapped onto the structural model, as shown in the plot to the right. A bit simplified, one can view the Cartesian grid model as representing the rock body at geological 'time zero' when the sediments have been deposited and have formed a stack of horizontal grid layers. From geological time zero and up to now, geological activity has introduced faults and deformed the layers, resulting in the structural model seen in the left plot of Figure 2.13.

Having seen the structural model, we continue to study the petrophysical parameters. The grid cells in our model are thought to be larger than the laminae of our imaginary reservoir and hence each grid block will generally contain both reservoir rock (with sufficient permeability) and impermeable shale. This is modelled using the net-to-gross ratio, `rock.ntg`, which is shown in Figure 2.14 along with the horizontal and vertical permeability. The plotting routines are exactly the same as for the porosity in Figure 2.13, but with different data and slightly different specification of the colorbar. From the figure, we clearly see that the model has a large content of shale and thus low permeability along the top. However, we also see high-permeable sand bodies that cut through the low-permeable top. In general, the permeabilities seem to correlate well with the sand content given by the net-to-gross parameter.

Some parts of the sand bodies are partially covered by mud that strongly reduces the vertical communication, most likely because of flooding events. These mud-draped surfaces occur on a sub-grid scale and are therefore modelled through a multiplier value (`MULTZ`) between zero and one that can be used to manipulate the effective communication (the transmissibility) between a given cell (i, j, k) and the cell immediately above $(i, j, k + 1)$. For complete-

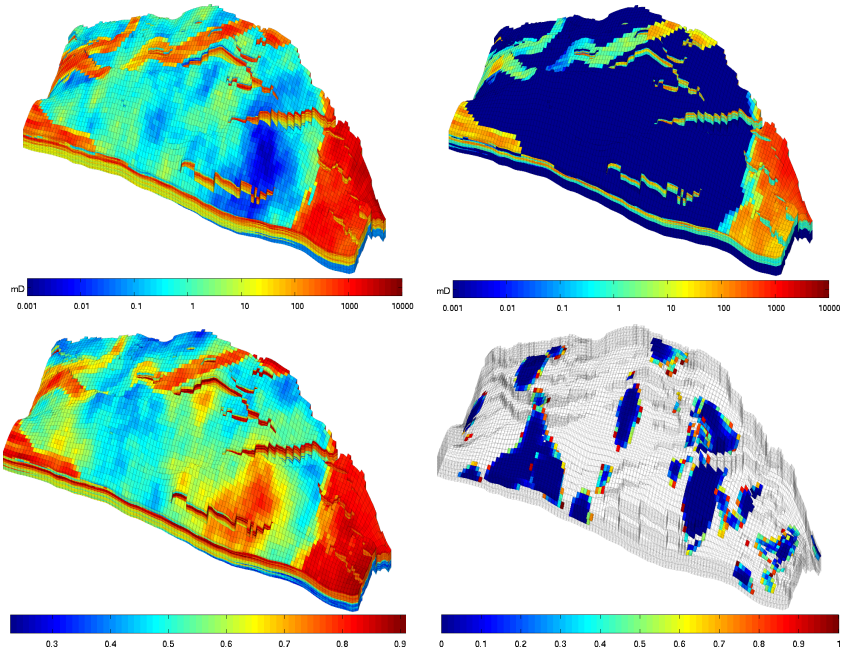


Fig. 2.14. The upper plots show the horizontal (left) and vertical permeability (right) for the SAIGUP model, using a logarithmic color scale. The lower plots show net-to-gross (left) and vertical multiplier values less than unity (right).

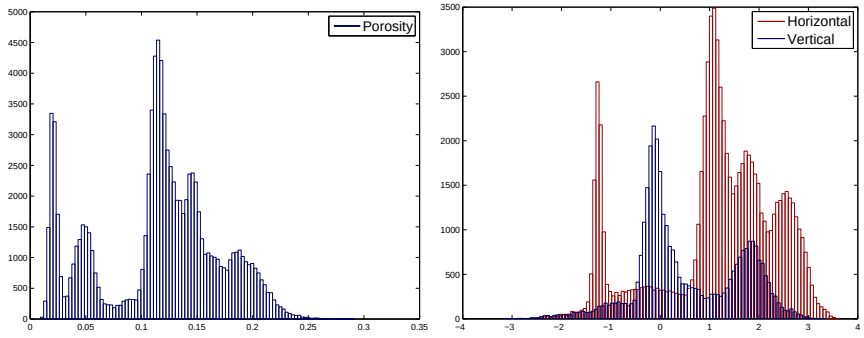


Fig. 2.15. Histogram of the porosity (left) and the logarithm of the horizontal and vertical permeability (right).

ness, we remark that the horizontal multiplier values (`MULTX` and `MULTY`) play a similar role for vertical faces, but are equal one in (almost) all cells for this particular realization.

To further investigate the heterogeneity of the model, we next look at histograms of the porosity and the permeabilities, as we did for the SPE 10 example (the MATLAB statements are omitted since they are almost iden-

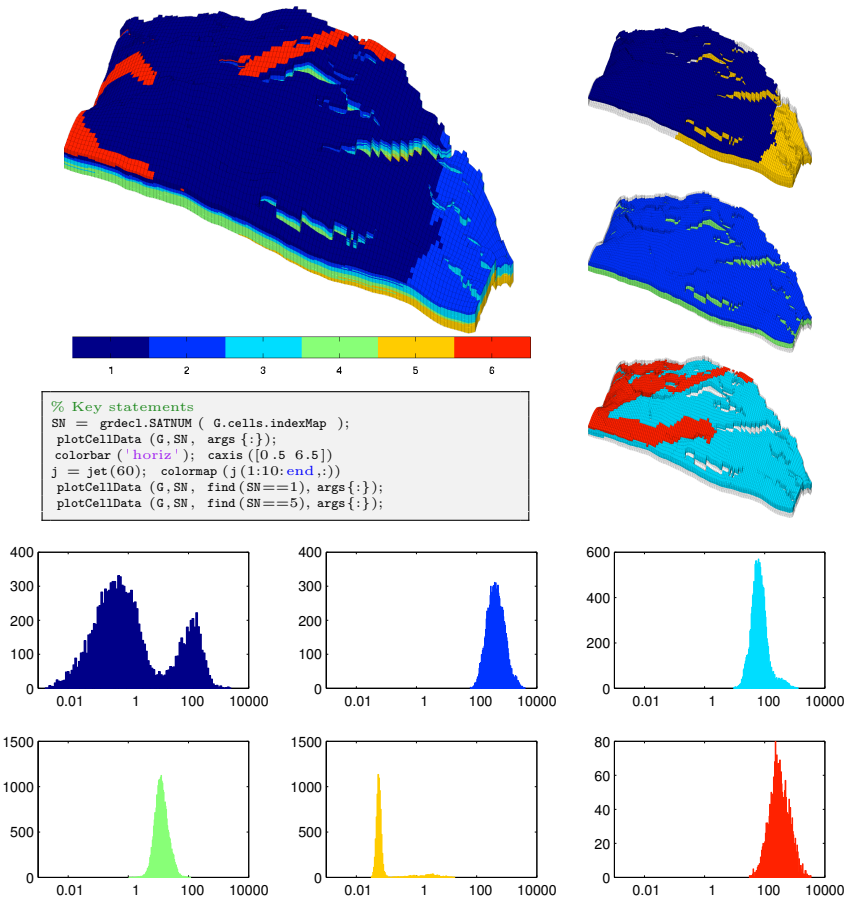


Fig. 2.16. The upper-left plot shows the rock type distribution for the SAIGUP model. The right column shows the six rock types grouped in pairs; from top to bottom, rock types number 1 and 5, 2 and 4, and 3 and 6. The bottom part of the figure shows histograms of the lateral permeability in units [mD] for each of the six rock types found in the SAIGUP model.

tical). In Figure 2.15, we clearly see that the distributions of porosity and horizontal permeability are multi-modal in the sense that five different modes can be distinguished, corresponding to the five different facies used in the petrophysical modelling.

It is common modelling practice that different rock types are assigned different rock-fluid properties (relative permeability and capillary functions), more details about such properties will be given later in the book. In the Eclipse input format, these different rock types are represented using the SATNUM keyword. By inspection of the SATNUM field in the input data, we see that the model contains six different rock types as depicted in Figure 2.16.

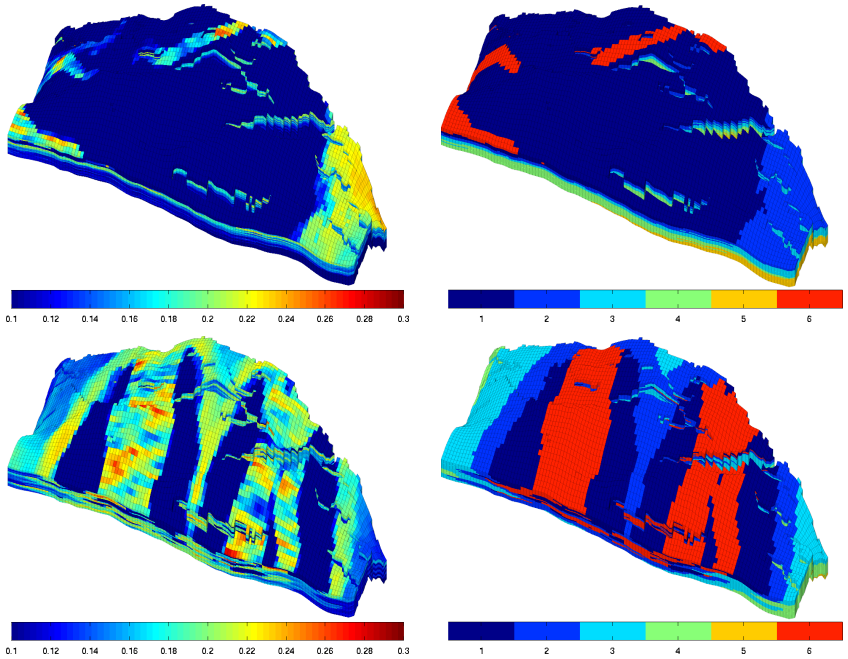


Fig. 2.17. Comparison of porosity (left) and the distribution of rock types (right) for two different SAIGUP realizations.

For completeness, the figure also shows the permeability distribution inside each rock type. Interestingly, the permeability distribution is multi-modal for at least two of the rock types.

Finally, to demonstrate the large difference in heterogeneity resulting from different depositional environment, we compare the realization we have studied above with another realization. In Figure 2.17 we show porosities and rock-type distributions. Whereas our original realization seems to correspond to a depositional environment with a flat shoreline, the other realization corresponds to a two-lobed shoreline, giving distinctively different facies belts. The figure also clearly demonstrates how the porosity (which depends on the grain-size distribution and packing) varies with the rock types. This can be confirmed by a quick analysis:

```
for i=1:6, pavg(i) = mean(rock.poro(SN==i));
           navg(i) = mean(rock.ntg(SN==i)); end
```

```
pavg = 0.0615    0.1883    0.1462    0.1145    0.0237    0.1924
navg = 0.5555    0.8421    0.7554    0.6179    0.3888    0.7793
```

In other words, rock types two and six are good sands with high porosity, three and four have intermediate porosity, whereas one and five correspond to less quality sand with a high clay content and hence low porosity.

3

Grids in Subsurface Modeling

The basic geological description of a petroleum reservoir or an aquifer system will typically consist of two sets of surfaces. Geological horizons are lateral surfaces that describe the bedding planes that delimit the rock strata, whereas faults are vertical or inclined surfaces along which the strata may have been displaced by geological processes. In this chapter, we will discuss how to turn the basic geological description into a discrete model that can be used to formulate various computational methods, e.g., for solving the equations that describe fluid flow.

A *grid* is a tessellation of a planar or volumetric object by a set of contiguous simple shapes referred to as *cells*. Grids can be described and distinguished by their *geometry*, reflected by the shape of the cells that form the grid, and their *topology* that tells how the individual cells are connected. In 2D, a cell is in general a closed polygon for which the geometry is defined by a set of *vertices* and a set of *edges* that connect pairs of vertices and define the interface between two neighboring cells. In 3D, a cell is a closed polyhedron for which the geometry is defined by a set of vertices, a set of edges that connect pairs of vertices, and a set of *faces* (surfaces delimited by a subset of the edges) that define the interface between two different cells, see Figure 3.1. Herein, we will assume that all cells in a grid are non-overlapping, so that each point in the planar/volumetric object represented by the grid is either inside a single cell, lies on an interface or edge, or is a vertex. Two cells that share a common face are said to be connected. Likewise, one can also define connections based on edges and vertices. The topology of a grid is defined by the total set of connections, which is sometimes also called the *connectivity* of the grid.

When implementing grids in modeling software, one always has the choice between generality and efficiency. To represent an arbitrary grid, it is necessary to explicitly store the geometry of each cell in terms of vertices, edges, and faces, as well as storing the connectivity among cells, faces, edges, and vertices. However, as we will see later, huge simplifications can be made for particular classes of grids by exploiting regularity in the geometry and structures in the topology. Consider, for instance, a planar grid consisting of rectangular

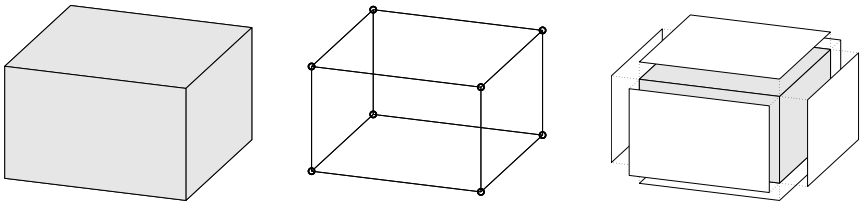


Fig. 3.1. Illustration of a single cell (left), vertices and edges (middle), and cell faces (right).

cells of equal size. Here, the topology can be represented by two indices and one only needs to specify a reference point and the two side lengths of the rectangle to describe the geometry. This way, one ensures minimal memory usage and optimal efficiency when accessing the grid. On the other hand, exploiting the simplified description explicitly in your flow or transport solver inevitably means that the solver must be reimplemented if you later decide to use another grid format.

The most important goal for our development of MRST is to provide a toolbox that both allows *and* enables the use of various grid types. To avoid having a large number of different, and potentially incompatible, grid representations, we have therefore chosen to store *all grid types* using a general unstructured format in which cells, faces, vertices, and connections between cells and faces are explicitly represented. This means that we, for the sake of generality, have sacrificed some of the efficiency one can obtain by exploiting special structures in a particular grid type and instead have focused on obtaining a flexible grid description that is not overly inefficient. Moreover, our grid structure can be extended by other properties that are required by various discretization schemes for flow and transport simulations. A particular discretization may need the *volume* or the *centroid* (grid-point, midpoint, or generating point) of each cell. Likewise, for cell faces one may need to know the face areas, the face normals, and the face centroids. Although these properties can be computed from the geometry (and topology) of the grid, it is often useful to precompute and include them explicitly in the grid representation.

The first third of this chapter is devoted to standard grid formats that are available in MRST. We introduce examples of structured grids, including regular Cartesian, rectilinear, and curvilinear grids, and briefly discuss unstructured grids, including Delaunay triangulations and Voronoi grids. The purpose of our discussion is to demonstrate the basic grid functionality in MRST and show some key principles that can be used to implement new structured and unstructured grid formats. In the second part of the chapter, we discuss industry-standard grid formats for stratigraphic grids that are based on extrusion of 2D shapes (corner-point, prismatic, and 2.5D PEBI grids). Although these grids have an inherent logical structure, representation of faults, erosion, pinch-outs, etc, leads to cells that can have quite irregular

shapes and an (almost) arbitrary number of faces. In the last part of the chapter, we discuss how the grids introduced in the first two parts of the chapter can be partitioned to form flexible coarse descriptions that preserve the geometry of the underlying fine grids. The ability to represent a wide range of grids, structured or unstructured on the fine and/or coarse scale, is a strength of MRST compared to the majority of research codes arising from academic institutions.

3.1 Structured Grids

As we saw above, a grid is a tessellation of a planar or volumetric object by a set of simple shapes. In a *structured* grid, only one basic shape is allowed and this basic shape is laid out in a regular repeating pattern so that the topology of the grid is constant in space. The most typical structured grids are based on quadrilaterals in 2D and hexahedrons in 3D, but in principle it is also possible to construct grids with a fixed topology using certain other shapes. Structured grids can be generalized to so-called multiblock grids (or hybrid grids), in which each block consists of basic shapes that are laid out in a regular repeating pattern.

Regular Cartesian grids

The simplest form of a structured grid consists of unit squares in 2D and unit cubes in 3D, so that all vertices in the grid are integer points. More generally, a regular Cartesian grid can be defined as consisting of congruent rectangles in 2D and rectilinear parallelepipeds in 3D, etc. Hence, the vertices have coordinates $(i_1\Delta x_1, i_2\Delta x_2, \dots)$ and the cells can be referenced using the multi-index $(i_1, i_2, \dots)$. Herein, we will only consider finite Cartesian grids that consist of a finite number $n_2 \times n_2 \times \cdots \times n_k$ of cells that cover a bounded domain $[0, L_1] \times [0, L_2] \times \cdots \times [0, L_k]$.

Regular Cartesian grids can be represented very compactly by storing n_i and L_i for each dimension. In MRST, however, Cartesian grids are represented as if they were fully unstructured using a general grid structure that will be described in more detail in Section 3.4. Cartesian grids therefore have special constructors,

```
G = cartGrid([nx, ny], [Lx Ly]);
G = cartGrid([nx, ny, nz], [Lx Ly Lz]);
```

that set up the data structures representing the basic geometry and topology of the grid. The second argument is optional.

Rectilinear grids

A rectilinear grid (also called a tensor grid) consists of rectilinear shapes (rectangles or parallelepipeds) that are not necessarily congruent to each other. In

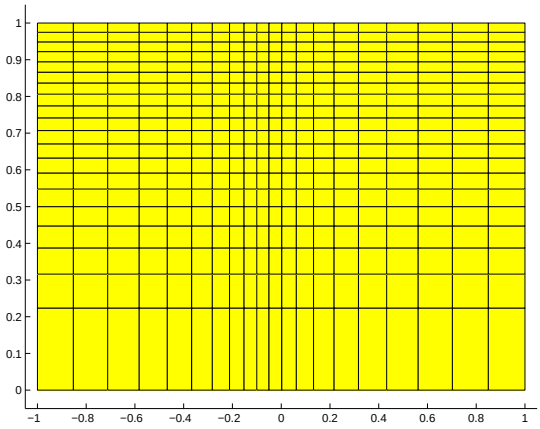


Fig. 3.2. Example of a rectilinear grid.

other words, whereas a regular Cartesian grid has a uniform spacing between its vertices, the grid spacing can vary along the coordinate directions in a rectilinear grid. The cells can still be referenced using a multi-index $(i_1, i_2, \dots)$ but the mapping from indices to vertex coordinates is nonuniform.

In MRST, one can construct a rectilinear grid by specifying the vectors with the grid vertices along the coordinate directions:

```
G = tensorGrid(x, y);
G = tensorGrid(x, y, z);
```

This syntax is the same as for the MATLAB functions `meshgrid` and `ndgrid`.

As an example of a rectilinear grid, we construct a 2D grid that covers the domain $[-1, 1] \times [0, 1]$ and is graded toward $x = 0$ and $y = 1$ as shown in Figure 3.2.

```
dx = 1-0.5*cos((-1:0.1:1)*pi);
x = -1.15+0.1*cumsum(dx);
y = 0:0.05:1;
G = tensorGrid(x, sqrt(y));
plotGrid(G); axis([-1.05 1.05 -0.05 1.05]);
```

Curvilinear grids

A curvilinear grid is a grid with the same topological structure as a regular Cartesian grid, but in which the cells are quadrilaterals rather than rectangles in 2D and cuboids rather than parallelepipeds in 3D. The grid is given by the coordinates of the vertices but there exists a mapping that will transform the curvilinear grid to a uniform Cartesian grid so that each cell can still be referenced using a multi-index $(i_1, i_2, \dots)$.

For the time being, MRST has no constructor for curvilinear grids. Instead, the user can create curvilinear grids by first instantiating a regular Cartesian or a rectilinear grid and then manipulating the vertices, as we will demonstrate next. This method is quite simple as long as there is a one-to-one mapping between the curvilinear grid in physical space and the logically Cartesian grid in reference space. The method will *not* work if the mapping is not one-to-one so that vertices with different indices coincide in physical space. In this case, the user should create an Eclipse input file with keywords `COORD[XYZ]`, see Section 3.3.1, and use the function `buildCoordGrid` to create the grid.

To illustrate the discussion, we show two examples of how to create curvilinear grids. In the first example, we create a rough grid by perturbing all internal nodes of a regular Cartesian grid (see Figure 3.3):

```
nx = 6; ny=12;
G = cartGrid([nx, ny]);
subplot(1,2,1); plotGrid(G);
c = G.nodes.coords;
I = any(c==0,2) | any(c(:,1)==nx,2) | any(c(:,2)==ny,2);
G.nodes.coords(~I,:) = c(~I,:) + 0.6*rand(sum(~I),2)-0.3;
subplot(1,2,2); plotGrid(G);
```

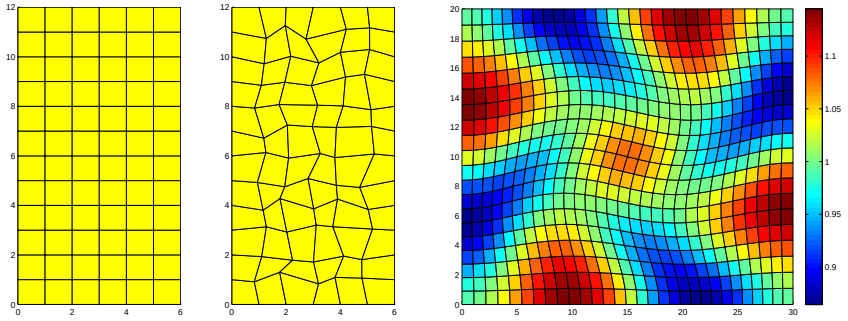


Fig. 3.3. The middle plot shows a rough grid created by perturbing all internal nodes of the regular 6×12 Cartesian grid in the left plot. The right plot shows a curvilinear grid created using the function `twister` that uses a combination of sin functions to perturbe a rectilinear grid. The color is determined by the cell volumes.

In the second example, we use the MRST example routine `twister` to perturb the internal vertices. The function maps the grid back to the unit square, perturbs the vertices according to the mapping

$$(x_i, y_i) \mapsto \big(x_i + f(x_i, y_i), y_i - f(x_i, y_i)\big), \quad f(x, y) = 0.03 \sin(\pi x) \sin\big(3\pi(y - \tfrac{1}{2})\big),$$

and then maps the grid back to its original domain. The resulting grid is shown in the right plot of Figure 3.3. To illuminate the effect of the mapping,

we have colored the cells according to their volume, which has been computed using the function `computeGeometry`, which we will come back to below.

```
G = cartGrid([30, 20]);
G.nodes.coords = twister(G.nodes.coords);
G = computeGeometry(G);
plotCellData(G, G.cells.volumes, 'EdgeColor', 'k'), colorbar
```

Fictitious domains

One obvious drawback with Cartesian and rectilinear grids, as defined above, is that they can only represent rectangular domains in 2D and cubic domains in 3D. Curvilinear grids, on the other hand, can represent more general shapes by introducing an appropriate mapping, and can be used in combination with rectangular/cubic grids in multiblock grids for efficient representation of realistic reservoir geometries. However, finding a mapping that conforms to a given boundary is often difficult, in particular for complex geologies, and using a mapping in the interior of the domain will inadvertently lead to cells with rough geometries that deviate far from being rectilinear. Such cells may in turn introduce problems if the grid is to be used in a subsequent numerical discretization, as we will see later.

As an alternative, complex geometries can be easily modelled using structured grids by a so-called fictitious domain method. In this method, the complex domain is embedded into a larger "fictitious" domain of simple shape (a rectangle or cube) using, e.g., a boolean indicator value in each cell to tell whether the cell is part of the domain or not. The observant reader will notice that we already have encountered the use of this technique for the SAIGUP dataset (Figure 2.12) and the Johansen dataset in Chapter 2. In some cases, one can also adapt the structured grid by moving the nearest vertices to the domain boundary.

MRST has support for fictitious domain methods through the function `removeCells`, which we will demonstrate in the next example, where we create a regular Cartesian grid that fills the volume of an ellipsoid:

```
x = linspace(-2,2,21);
G = tensorGrid(x,x,x);
subplot(1,2,1); plotGrid(G);view(3); axis equal

subplot(1,2,2); plotGrid(G,'FaceColor','none');
G = computeGeometry(G);
c = G.cells.centroids;
r = c(:,1).^2 + 0.25*c(:,2).^2+0.25*c(:,3).^2;
G = removeCells(G, r>1);
plotGrid(G); view(-70,70); axis equal;
```

Worth observing here is the use of `computeGeometry` to compute cell centroids which are not part of the basic geometry representation in MRST.

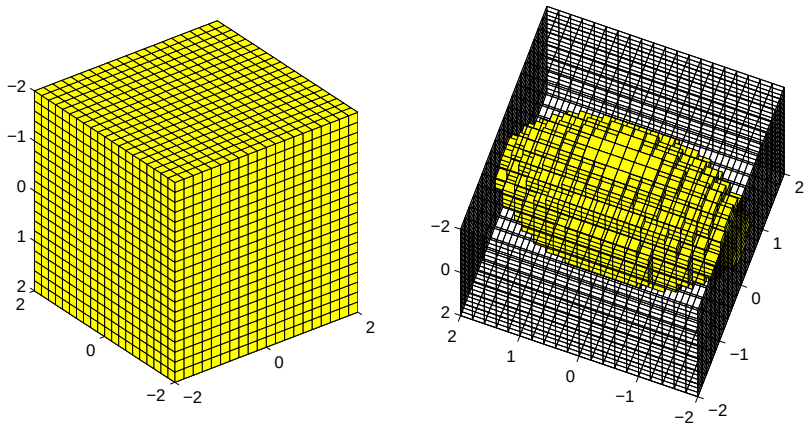


Fig. 3.4. Example of a regular Cartesian grid representing a domain in the form of an ellipsoid. The underlying logical Cartesian grid is shown in the left plot and as a wireframe in the right plot. The active part of the model is shown in yellow color in the right plot.

Plots of the grid before and after removing the inactive parts are shown in Figure 3.4. Because of the fully unstructured representation used in MRST, calling `computeGeometry` actually removes the inactive cells from the grid structure, but from the outside, the structure behaves as if we had used a fictitious domain method.

3.2 Unstructured Grids

An unstructured grid consists of a set of simple shapes that are laid out in an irregular pattern so that any number of cells can meet at a single vertex. The topology of the grid will therefore change throughout space. An unstructured grid can generally consist of a combination of polyhedral cells with varying number of faces, as we will see below. However, the most common forms of unstructured grids are based on triangles in 2D and tetrahedrons in 3D. These grids are very flexible and are relatively easy to adapt to complex domains and structures or refine to provide increased local resolution.

Unlike structured grids, unstructured grids cannot generally be efficiently referenced using a structured multi-index. Instead, one must describe a list of connectivities that specifies the way a given set of vertices make up individual element and element faces, and how these elements are connected to each other via faces, edges, and vertices.

To understand the properties and construction of unstructured grids, we start by a brief discussion of two concepts from computational geometry: Delaunay tessellation and Voronoi diagrams. Both these concepts are supported by standard functionality in MATLAB.

3.2.1 Delaunay Tessellation

A tessellation of a set of generating points $\mathcal{P} = \{x_i\}_{i=1}^n$ is defined as a set of simplices that completely fills the convex hull of $\mathcal{P}$. The convex hull H of $\mathcal{P}$ is the convex minimal set that contains $\mathcal{P}$ and can be described constructively as the set of convex combinations of a finite subset of points from $\mathcal{P}$,

$$H(\mathcal{P}) = \Big\{ \sum_{i=1}^{\ell} \lambda_i x_i \;\Big|\; x_i \in \mathcal{P}, \lambda_i \in \mathbb{R}, \lambda_i \geq 0, \sum_{i=1}^{\ell} \lambda_i = 1, 1 \leq \ell \leq n \Big\}.$$

Delaunay tessellation is by far the most common method of generating a tessellation based on a set of generating points. In 2D, the Delaunay tessellation consists of a set of triangles defined so that three points form the corners of a Delaunay triangle only when the circumcircle that passes through them contains no other points, see Figure 3.5. The definition using circumcircles can readily be generalized to higher dimensions using simplices and hyperspheres.

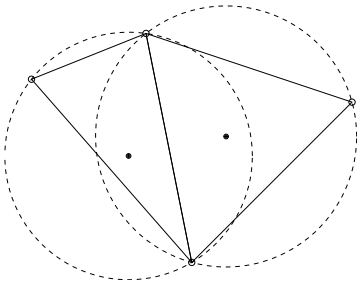


Fig. 3.5. Two triangles and their circumcircles.

The center of the circumcircle is called the circumcenter of the triangle. We will come back to this quantity when discussing Voronoi diagrams in the next subsection. When four (or more) points lie on the same circle, the Delaunay triangulation is not unique. As an example, consider four points defining a rectangle. Using either of the two diagonals will give two triangles satisfying the Delaunay condition.

The Delaunay triangulation can alternatively be defined using the so-called max-min angle criterion, which states that the Delaunay triangulation is the one that maximizes the minimum angle of all angles in a triangulation, see Figure 3.6. Likewise, the Delaunay triangulation minimizes the largest circumcircle and minimizes the largest min-containment circle, which is the smallest circle that contains a given triangle. Additionally, the closest two generating points are connected by an edge of a Delaunay triangulation. This is called the closest-pair property, and such two neighboring points are often referred to as natural neighbors. This way, the Delaunay triangulation can be seen as the natural tessellation of a set of generating points.

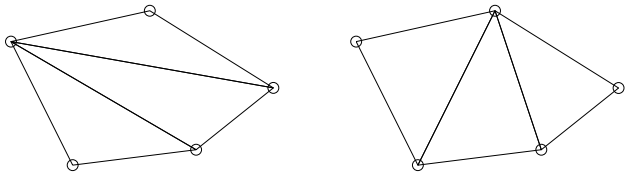


Fig. 3.6. Example of two triangulations of the same five points; the triangulation to the right satisfies the min-max criterion.

Delaunay tessellation is a popular research topic and there exists a large body of literature on theoretical aspects and computer algorithms. Likewise, there are a large number of software implementations available on the net. For this reason, MRST does not have any routines for generating tessellations based on simplexes. Instead, we have provided simple routines for mapping a set of points and edges, as generated by MATLAB's Delaunay triangulation routines, to the internal data structure used to represent grids in MRST. How they work, will be illustrated in terms of a few simple examples.

In the first example, we use routines from MATLAB's `polyfun` toolbox to triangulate a rectangular mesh and convert the result using the MRST routine `triangleGrid`:

```
[x,y] = meshgrid(1:10,1:8);
t = delaunay(x(:),y(:));
G = triangleGrid([x(:) y(:)],t);
plot(x(:),y(:),'o','MarkerSize',8);
plotGrid(G,'FaceColor','none');
```

Depending on what version you have of MATLAB, the 2D Delaunay routine `delaunay` will produce one of the triangulations shown in Figure 3.7. In older versions of MATLAB, the implementation of `delaunay` was based on 'QHULL' (see http://www.qhull.org), which produces the unstructured triangulation shown in the right plot. MATLAB 7.9 and newer has improved routines for 2-D and 3-D computational geometry, and here `delaunay` will produce the structured triangulation shown in the left plot. However, the n-D tessellation

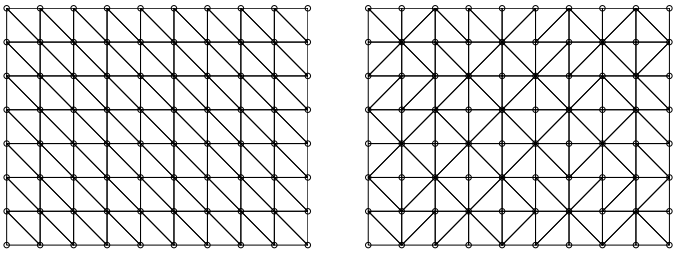


Fig. 3.7. Two different Delaunay tessellations of a rectangular point mesh.

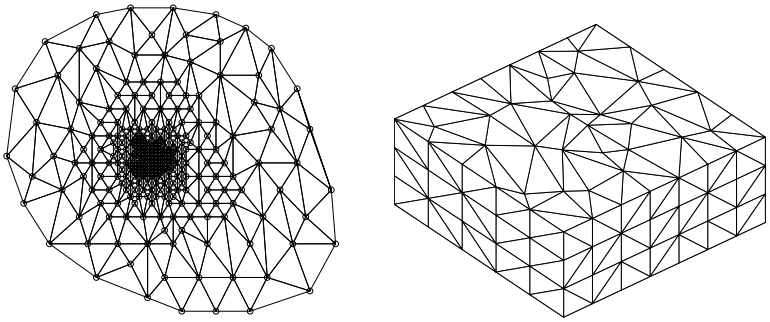


Fig. 3.8. The left plot shows the triangular grid from the `seamount` demo case. The right plot shows a tetrahedral tessellation of a 3D point mesh.

routine `delaunayn([x(:) y(:)])` is still based on 'QHULL' and will generally produce an unstructured tessellation, as shown in the right plot.

If the set of generating points is structured, e.g., as one would obtain by calling either `meshgrid` or `ndgrid`, it is straightforward to make a structured triangulation. The following skeleton of a function makes a 2D triangulation and can easily be extended by the interested reader to 3D:

```
function t = mesh2tri(n,m)
[I,J]=ndgrid(1:n-1, 1:m-1); p1=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(2:n , 1:m-1); p2=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(1:n-1, 2:m ); p3=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(2:n , 1:m-1); p4=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(2:n , 2:m ); p5=sub2ind([n,m],I(:),J(:));
[I,J]=ndgrid(1:n-1, 2:m ); p6=sub2ind([n,m],I(:),J(:));
t = [p1 p2 p3; p4 p5 p6];
```

In Figure 3.8, we have used the demo case `seamount` that is supplied with MATLAB as an example of a more complex unstructured grid

```
load seamount;
plot(x(:),y(:), 'o');
G = triangleGrid([x(:) y(:)]);
plotGrid(G,'FaceColor',[.8 .8 .8]); axis off;
```

The observant reader will notice that here we do not explicitly generate a triangulation before calling `triangleGrid`; if the second argument is omitted, the routine uses MATLAB's builtin `delaunay` triangulation as default.

For 3D grids, MRST supplies a conversion routine `tetrahedralGrid(P, T)` that constructs a valid grid definition from a set of points `P` ($m \times 3$ array of node coordinates) and a tetrahedron list `T` (n array of node indices). The tetrahedral tessellation shown to the right in Figure 3.8 was constructed from a set of generating points defined by perturbing a regular hexahedral point mesh:

```
N=7; M=5; K=3;
[x,y,z] = ndgrid(0:N,0:M,0:K);
x(2:N,2:M,:) = x(2:N,2:M,:) + 0.3*randn(N-1,M-1,K+1);
y(2:N,2:M,:) = y(2:N,2:M,:) + 0.3*randn(N-1,M-1,K+1);
G = tetrahedralGrid([x(:) y(:) z(:)]);
plotGrid(G, 'FaceColor',[.8 .8 .8]); view(-40,60); axis tight off
```

3.2.2 Voronoi Diagrams

The Voronoi diagram of a set of points $\mathcal{P} = \{x_i\}_{i=1}^n$ is the partitioning of Euclidean space into n (possibly unbounded) convex polytopes[1] such that each polytope contains exactly one generating point x_i and every point inside the given polytope is closer to its generating point than any other point in $\mathcal{P}$. The convex polytopes are called Voronoi cells (or Voronoi regions). Mathematically, the Voronoi cell $V(x_i)$ of generating point x_i in $\mathcal{P}$ can be defined as

$$V(x_i) = \Big\{ x \mid \|x - x_i\| < \|x - x_j\| \; \forall j \neq i \Big\}. \tag{3.1}$$

A Voronoi region is not closed in the sense that a point that is equally close to two or more generating points does not belong to the region defined by (3.1). Instead, these points are said to lie on the Voronoi segments and can be included in the Voronoi cells by defining the closure of $V(x_i)$, using "$\leq$" rather than "$<$" in (3.1).

The Voronoi cells for all generating points lying at the convex hull of $\mathcal{P}$ are unbounded, all other Voronoi cells are bounded. For each pair of two points x_i and x_j, one can define a hyperplane with co-dimension one consisting of all points that lie equally close to x_i and x_j. This hyperplane is the perpendicular bisector to the line segment between x_i and x_j and passes through the midpoint of the line segment. The Voronoi diagram of a set of points can be derived directly as the *dual* of the Delaunay triangulation of the same points. To understand this, we consider the planar case, see Figure 3.9. For every triangle, there is a polyhedron in which vertices occupy complementary locations:

- The circumcenter of a Delaunay triangle corresponds to a vertex of a Voronoi cell.
- Each vertex in the Delaunay triangulation corresponds to, and is the center of, a Voronoi cell.

Moreover, for locally orthogonal Voronoi diagrams, an edge in the Delaunay triangulation corresponds to a segment in the Voronoi diagram and the two intersect each other orthogonally. However, as we can see in Figure 3.9, this

[1] A polytope is a generic term that refers to a polygon in 2D, a polyhedron in 3D, etc

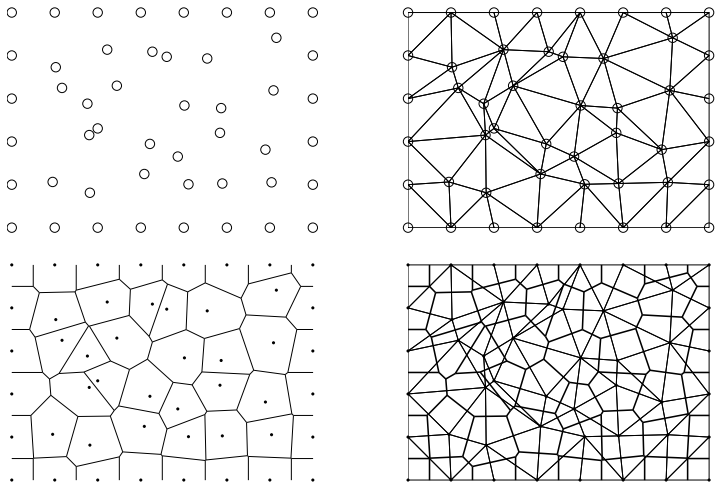


Fig. 3.9. Duality between Voronoi diagrams and Delaunay triangulation. From top left to bottom right: generating points, Delaunay triangulation, Voronoi diagram, and Voronoi diagram (thick lines) and Delaunay triangulation (thin lines).

is not always the case. If the circumcenter of a triangle lies outside the triangle itself, the Voronoi segment does not intersect the corresponding Delaunay edge. To avoid this situation, one can perform a constrained Delaunay triangulation and insert additional points where the constraint is not met (i.e., the circumcenter is outside its triangle).

Figure 3.10 shows three examples of planar Voronoi diagrams generated from 2D point lattices using the MATLAB-function `voronoi`. MRST has not yet a similar function that generates a Voronoi grid from a point set, but offers instead `V=pebi(T)` that generates a locally orthogonal, 2D Voronoi grid `V` as a dual to a triangular grid `T`. The grids are constructed by connecting the perpendicular bisectors of the edges of the Delaunay triangulation, hence the name PErpendicular BIsector (PEBI) grids. To demonstrate the functionality, we first consider the honeycombed grids to the left in Figure 3.10

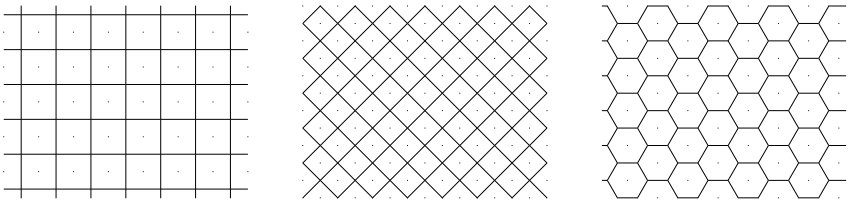


Fig. 3.10. Three examples of Voronoi diagrams generated from 2D point lattices. From left to right: square lattice, square lattice rotated 45 degrees, lattice forming equilateral triangles.

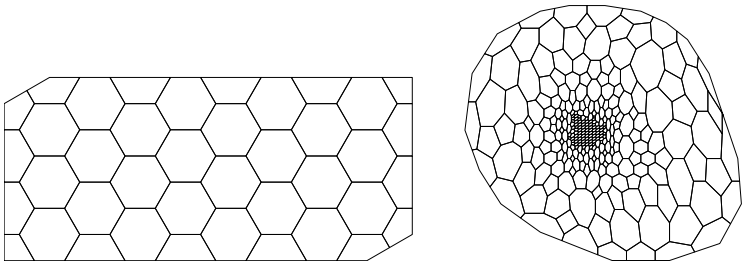


Fig. 3.11. Two examples of Voronoi grids. The left plot shows a honeycombed PEBI grid and the right plot shows the PEBI grid derived from the `seamount` demo case.

```
[x,y] = meshgrid([0:4]*2*cos(pi/6),0:3);
x = [x(:); x(:)+cos(pi/6)];
y = [y(:); y(:)+sin(pi/6)];
G = triangleGrid([x,y]);
plotGrid(pebi(G), 'FaceColor','none'); axis equal off
```

The result is shown in Figure 3.11. As a second example, we reiterate the `seamount` examples shown in Figure 3.8 is straightforward

```
load seamount
V = pebi(triangleGrid([x y]));
plotGrid(V,'FaceColor',[.8 .8 .8]); axis off;
```

Since `pebi` is a 2D code, we cannot apply it directly[2] to the 3D tetrahedral grid shown in Figure 3.8 to generate a dual 3D Voronoi grid. As we will see in the next section, a more feasible approach in geological modeling is to construct a 2D Voronoi grid which can be extruded to 3D to preserve geological layering.

3.3 Stratigraphic Grids

In the previous chapter, we saw that grid models are used as an important ingredient in describing the geometrical and petrophysical properties of a subsurface reservoir. This means that the grid is closely attached to the parameter description of the flow model and, unlike in many other disciplines, cannot be chosen arbitrarily to provide a certain numerical accuracy. Indeed, the grid is typically chosen by a geologist who tries to describe the rock body by as few volumetric cells as possible and who basically does not care too much about potential numerical difficulties his or her choice of grid may cause in

[2] Extending `pebi` to generate 3D Voronoi grid is probably an example of what typically ends up as exercises in certain textbooks, i.e., something the authors believe is possible to do but have actually not bothered to verify themselves.

subsequent flow simulations. This statement is, of course, grossly simplified but is important to bear in mind throughout the rest of this chapter.

The industry standard for representing the reservoir geology in a flow simulator is through the use of a stratigraphic grid that is built based on geological horizons and fault surfaces. The volumetric grid is typically built by extruding 2D tessellations of the geological horizons in the vertical direction or in a direction following major fault surfaces. For this reason, some stratigraphic grids, like the PEBI grids that we will meet in Section 3.3.2, are often called 2.5D rather than 3D grids. These grids may be unstructured in the lateral direction, but have a clear structure in the vertical direction to reflect the layering of the reservoir.

Because of the role grid models play in representing geological formations, real-life stratigraphic grids tend to be highly complex and have unstructured connections induced by the displacements that have occured over faults. Another characteristic feature is high aspect ratios. Typical reservoirs extend several hundred or thousand meters in the lateral direction, but the zones carrying hydrocarbon may be just a few tens of meters in the vertical direction and consist of several layers with (largely) different rock properties. Getting the stratigraphy correct is crucial, and high-resolution geological modeling will typically result in a high number of (very) thin grid layers in the vertical direction, resulting in two or three orders of magnitude aspect ratios.

A full exposition of stratigraphic grids is way beyond the scope of this book. In next two subsections, we will discuss the basics of the two most commonly used forms of stratigraphic grids.

3.3.1 Corner-Point Grids

To model the geological structures of petroleum reservoirs, the industry-standard approach is to introduce what is called a corner-point grid [53], which we already encountered in Chapter 2.4. A corner-point grid consists of a set of hexahedral cells that are topologically aligned in a Cartesian fashion so that the cells can be numbered using a logical ijk index. In its simplest form, a corner-point grid is specified in terms of a set of vertical or inclined pillars defined over an areal Cartesian 2D mesh in the lateral direction. Each cell in the volumetric grid has eight *logical* corner points that are restricted by four pillars and specified as two depth-coordinates on each pillar, see Figure 3.12. Each grid consists of $n_x \times n_y \times n_z$ grid cells and the cells are ordered with the i-index (x-axis) cycling fastest, then the j-index (y-axis), and finally the k-index (negative z-direction). All cellwise property data are assumed to follow the same numbering scheme.

As discussed previously, a fictitious domain approach is used to embed the reservoir in a logically Cartesian shoe-box. This means that inactive cells that are not part of the physical model, e.g., as shown in Figure 2.12, are present in the topological ijk-numbering but are indicated by a zero porosity or net-

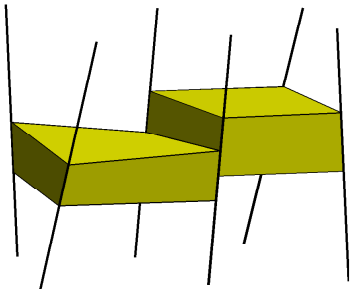


Fig. 3.12. Each cell in the corner-point grid is restricted by four pillars and two points on each pillar.

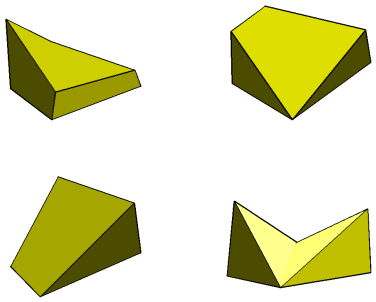


Fig. 3.13. Examples of deformed and degenerate hexahedral cells arising in corner-point grid models.

to-gross value, as discussed in Chapter 2.3 or marked by a special boolean indicator (called `ACTNUM` in the input files).

So far, the topology and geometry of a corner-point grid have not deviated from that of the mapped Cartesian grids studied in the previous section. Somewhat simplified, one may view the logical ijk numbering as a reflection of the sedimentary rock bodies as they may have appeared at geological 'time zero' when all rock facies have been deposited as part of horizontal layers in the grid (i.e., cells with varying i and j but constant k). To model geological features like erosion and pinch-outs of geological layers, the corner-point format allows point-pairs to collapse along pillars. This creates degenerate hexahedral cells that may have less than six faces, as illustrated in Figure 3.13. The corner points can even collapse along all four pillars, so that a cell completely disappears. This will implicitly introduce a new topology, which is sometimes referred to as 'non-neighboring connections', in which cells that are not logical k neighbors can be neighbors and share a common face in physical space. An example of a model that contains both eroded geological layers and fully collapsed cells is shown in Figure 3.14. In a similar manner, (simple) vertical and inclined faults can be easily modelled by aligning the pillars with fault

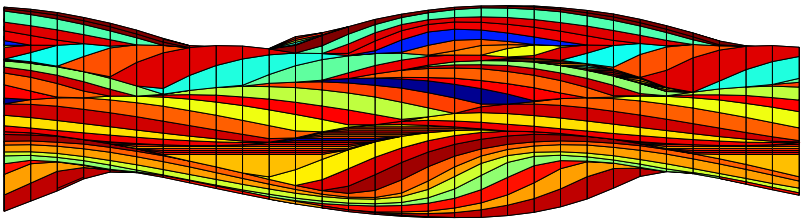


Fig. 3.14. Side view in the xz-plane of corner-point grid with vertical pillars modeling a stack of sedimentary beds (each layer indicated by a different color).

surfaces and displacing the corner points defining the neighboring cells on one or both sides of the fault. This way, one creates non-matching geometries and non-neighboring connections in the underlying ijk topology.

To illustrate the concepts introduced so far, we consider a low-resolution version of the model from Figure 2.7 created by the `simpleGrdecl` grid-factory routine, which generates the basic 'COORD' and 'ZCORN' properties that are used to describe a corner-point grid in the Eclipse input deck. We start by creating the input stream, from which we extract the pillars and the corner points:

```
grdecl = simpleGrdecl([4, 2, 3], .12, 'flat', true);
[X,Y,Z] = buildCornerPtPillars(grdecl,'Scale',true);
[x,y,z] = buildCornerPtNodes(grdecl);
```

The pillars are given in the `COORD` field and the vertical positions of the corner points along each pillar are given in the `ZCORN` field. Having obtained the necessary data, we plot the pillars and the corner-points and mark pillars on which the corner-points of logical ij neighbors do not coincide,

```
% Plot pillars
plot3(X',Y',Z','k');
set(gca,'zdir','reverse'), view(35,35), axis off, zoom(1.2);

% Plot points on pillars, mark pillars with faults red
hold on; I=[3 8 13];
hpr = plot3(X(I,:)',Y(I,:)', Z(I,:)', 'r','LineWidth',2);
hpt = plot3(x(:),y(:),z(:),'o'); hold off;
```

The resulting plots are shown in the upper row of Figure 3.15, in which we clearly see how the pillars change slope from the east and west side toward the fault in the middle, and how the grid points sit like beads-on-a-string along each pillar.

Cells are now defined by connecting pairs of points from four neighboring pillars that make up a rectangle in the lateral direction. To see this, we plot two vertical stacks of cells and finally the whole grid with the fault surface marked in blue:

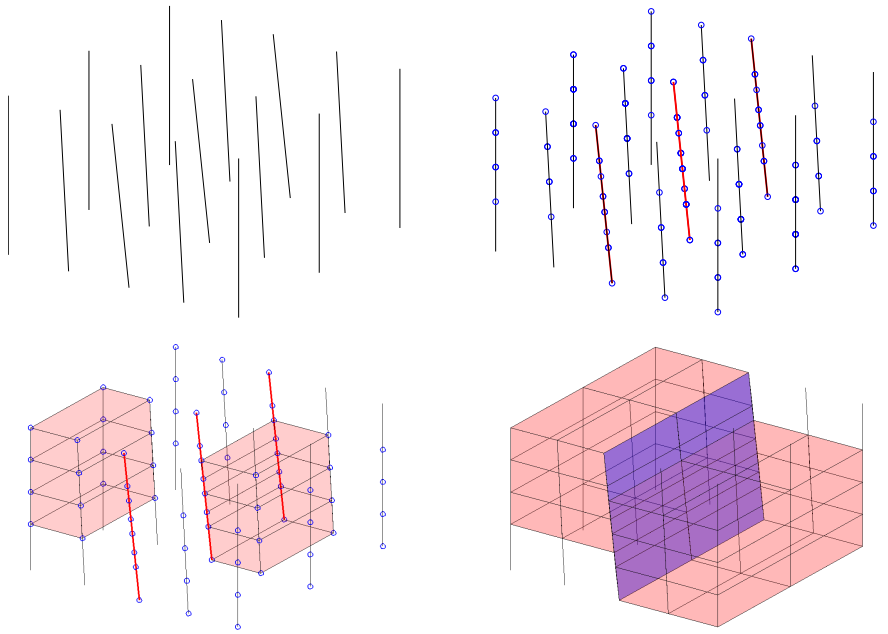


Fig. 3.15. Specification of a corner-point grid. Starting from the pillars (upper left), we add corner-points and identify pillars containing non-matching corner marked in red (upper right). A stack of cells is created for each set of four pillars (lower left), and then the full grid is obtained (lower right). In the last plot, the fault faces have been marked in blue.

```
% Create grid and plot two stacks of  cells
G = processGRDECL(grdecl);
args = {'FaceColor'; 'r';  'EdgeColor'; 'k'};
hcst = plotGrid(G,[1:8:24  7:8:24], 'FaceAlpha', .1, args{:});

% Plot  cells  and fault  surface
delete([hpt; hpr; hcst]);
plotGrid(G,'FaceAlpha', .15, args{:});
plotFaces(G, G.faces.tag>0,'FaceColor','b','FaceAlpha',.4);
```

The upper-left plot in Figure 3.16 shows the same model sampled with even fewer cells. To highlight the non-matching cell faces along the fault plane we have used different coloring of the cell faces on each side of the fault. In MRST, we have chosen to represent corner-point grids as *matching* unstructured grids obtained by subdividing all non-matching cell faces, instead of using the more compact non-matching hexahedral form. For the model in Figure 3.16, this means that the four cells that have non-neighboring connections across the fault plane will have seven and not six faces. For each such cell, two of the

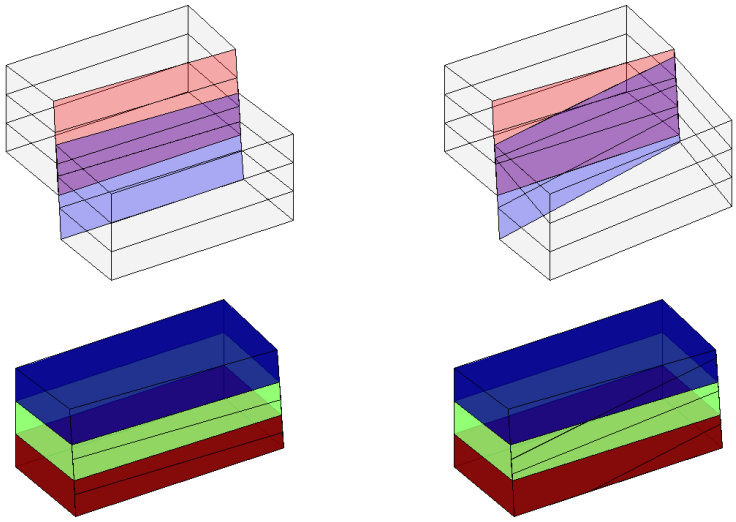


Fig. 3.16. Subdivision of fault face in two three-dimensional models. In the left column, the subfaces are all rectangular. In the right columns they are not. In both the upper plots, the faces marked in red belong only to the cells behind the fault surface, the blue faces belong only to the cells in front of the fault surface, and the magenta ones belong to cells on both sides. The lower plot shows the cells behind the surface, where each cell has been given its own color.

seven faces lie along the fault plane. For the regular model studied here, the subdivision results in new faces that all have four corners (and are rectangular). However, this is not generally the case, as is shown in the right column of Figure 3.16, where we can see cells with six, seven, eight faces, and faces with three, four, and five corners. Indeed, for real-life models, subdivision of non-matching fault faces can lead to cells that have much more than six faces.

Using the inherent flexibility of the corner-point format it is possible to construct very complex geological models that come a long way in matching the geologist's perception of the underlying rock formations. Because of their many appealing features, corner-point grids have been an industry standard for years and the format is supported in most commercial software for reservoir modeling and simulation.

A Synthetic Faulted Reservoir

In our first example, we consider a synthetic model of two intersecting faults that make up the letter Y in the lateral direction. The two fault surfaces are highly deviated, making an angle far from 90 degrees with the horizontal direct. To model this scenario using corner-point grids, we basically have two

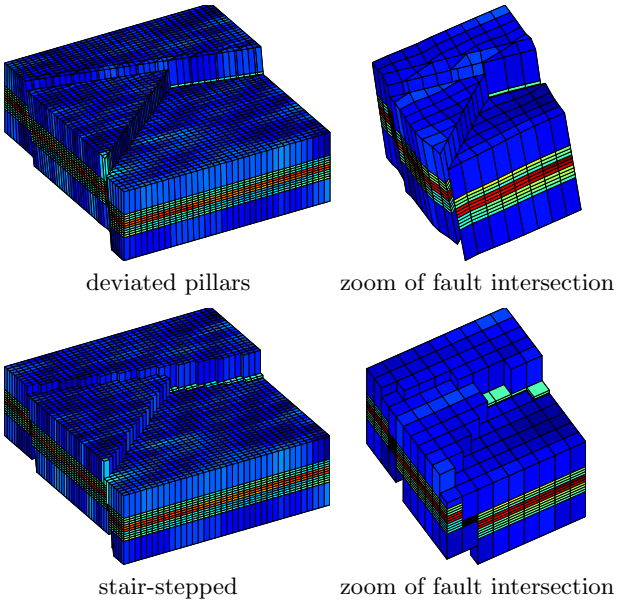


Fig. 3.17. Modeling the intersection of two deviated faults using deviated pillars (top) and stair-stepped approximation (bottom). Grids courtesy of Statoil.

different choices. The first choice, which is quite common, is to let the pillars (and hence the extrusion direction) follow the main fault surfaces. For highly deviated faults, like in the current case, this will lead to extruded cells that are far from K-orthogonal and hence susceptible to grid-orientation errors in a subsequent simulation, as will be discussed in more detail in Chapter 8. Alternatively, we can choose a vertical extrusion direction and replace deviated fault surfaces by stair-stepped approximations so that the faults zig-zag in direction not aligned with the grid. This will create cells that are mostly K-orthogonal and less prone to grid-orientation errors.

Figure 3.17 shows two different grid models, one with deviated pillars and one with stair-stepped faults. Whereas the latter has orthogonal cells, the former has cells with geometries degenerate or deviate strongly from being orthogonal in the lateral direction. Likewise, some pillars are close to 45 degrees inclination, which will likely give significant grid-orientation effects in a standard two-point scheme.

A Simulation Model of the Norne Field

Norne is an oil and gas field lies located in the Norwegian Sea. The reservoir is found in Jurrasic sandstone at a depth of 2500 meter below sea level, and was originally estimated to contain 90.8 million Sm^3 oil, mainly in the Ile and Tofte formations, and 12.0 billion Sm^3 in the Garn formation. The field

is operated by Statoil and production started in November 1997, using a floating production, storage and offloading (FPSO) ship connected to seven subsea templates at a water depth of 380 meters. The oil is produced with water injection as the main drive mechanisms and the expected ultimate oil recovery is more than 60%, which is very high for a subsea oil reservoir. During thirteen years of production, five 4D seismic surveys of high quality have been recorded. Operator Statoil and partners (ENI and Petoro) have agreed with NTNU to release large amounts of subsurface data from the Norne field for research and education purposes. An important objective of this agreement is to establish a number of international benchmark cases based on real data for the testing of reservoir characterization/history matching and/or production optimization methodologies. More specifically, the Norne Benchmark datasets are hosted and supported by the Center for Integrated Operations in the Petroleum Industry (IO Center) at NTNU:

http://www.ipt.ntnu.no/~norne/wiki/doku.php

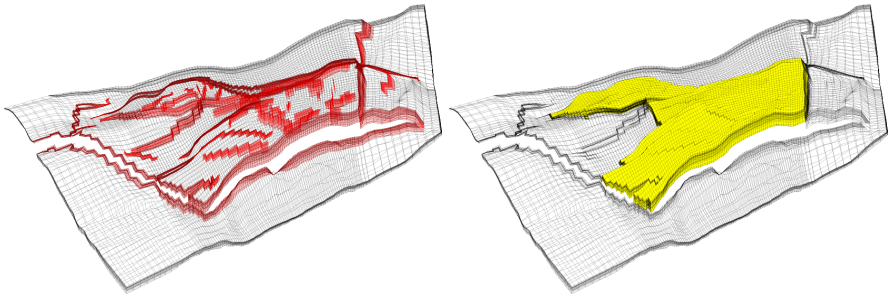
Here, we will use the simulation model released as part of "Package 2: Full field model" (2013) as an example of a real reservoir. We emphasize that the view expressed in the following are the views of the authors and do not necessarily reflect the views of Statoil and the Norne license partners.

The model consists of a $46 \times 112 \times 22$ corner-point grid, given in the Eclipse format, which can be read as discussed for the SAIGUP model in Chapter 2.4. We start by plotting the whole model, including inactive cells. To this end, we need to override[3] the `ACTNUM` field before we start processing the input, because if the `ACTNUM` flag is set, all inactive cells will be ignored when the unstructured grid is built

```
actnum = grdecl.ACTNUM;
grdecl.ACTNUM = ones(prod(grdecl.cartDims),1);
G  = processGRDECL(grdecl, 'checkgrid', false);
```

Having obtained the grid in the correct unstructured format, we first plot the outline of the whole model and highlight all faults and the active part of the model, see Figure 3.18. During the processing, all fault faces are tagged with a positive number. This can be utilized to highlight the faults: we simply find all faces with a positive tag, and color them with a specific color as shown in the left box in the figure. We now continue with the active model only. Hence, we reset the ACTNUM field to its original values so that inactive cells are ignored when we process the Eclipse input stream. In particular, we will examine some parts of the model in more detail. To this end, we will use the

[3] At this point we hasten to warn the reader that inactive cells often contain garbage data and may generally not be inspected in this manner. Here, however, most inactive cells are defined in a reasonable way. By not performing basic sanity checks on the resulting grid (option 'checkgrid'=`false`), we manage to process the grid and produce reasonable graphical output. In general, however, we strongly advice that 'checkgrid' remains set in its default state of `true`.



```
plotGrid(G,'FaceColor','none','EdgeAlpha',.1);
plotFaces(G, G.faces.tag >0, ...
   'FaceColor','red','FaceAlpha',.2, ...
   'EdgeColor','r','EdgeAlpha',.1);
axis off; view(-155,80); zoom(1.7);
```

```
plotGrid(G, ~actnum( G.cells.indexMap ), ...
   'FaceColor','none','EdgeAlpha',.1);
plotGrid(G, actnum( G.cells.indexMap ), ...
   'FaceColor','y', 'EdgeAlpha',.1);
axis off; view(-155,80); zoom(1.7);
```

Fig. 3.18. The Norne field, a real model from the Norwegian Sea. The plots show the whole grid with fault faces marked in red (left) and active cells marked in yellow (right).

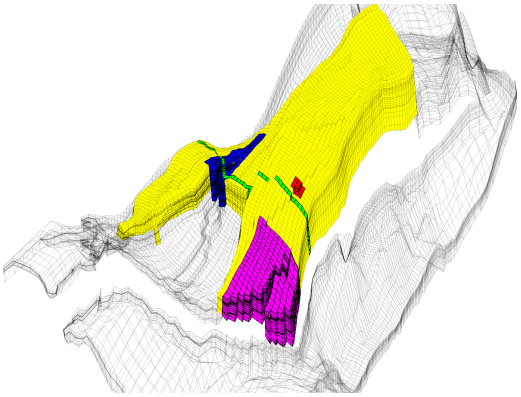
function `cutGrdecl` that extracts a rectangular box in index space from the Eclipse input stream, e.g., as follows

```
cut_grdecl = cutGrdecl(grdecl, [6 15; 80 100; 1 22]);
g = processGRDECL(cut_grdecl);
```

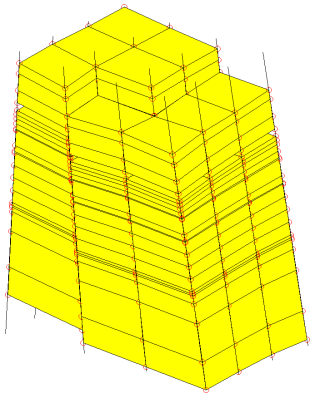
In Figure 3.19, we have zoomed in on four different regions. The first region (red color), is sampled near a laterally stair-stepped fault, which is a curved fault surface that has been approximated by a surface that zig-zags in the lateral direction. We also notice how the fault displacement leads to cells that are non-matching across the fault surface and the presence of some very thin layers (the thinest layers may actually appear to be thick lines in the plot). The thin layers are also clearly seen in the second region (magenta color), which represents a somewhat larger sample from an area near the tip of one of the 'fingers' in the model. Here, we clearly see how similar layers have been strongly displaced across the fault zone. In the third (blue) region, we have colored the fault faces to clearly show the displacement and the hole through the model in the vertical direction, which likely corresponds to a shale layer that has been eliminated from the active model. Gaps and holes, and displacement along fault faces, are even more evident for the vertical cross-section (green region) for which the layers have been given different colors as in Figure 3.14. Altogether, the four views of the model demonstrate typical patterns that can be seen in realistic models.

Extensions, Difficulties, and Challenges

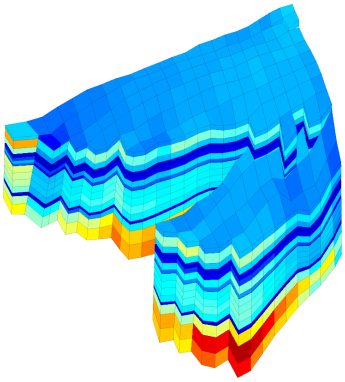
The original corner-point format has been extended in several directions, for instance to enable vertical intersection of two straight pillars in the shape of



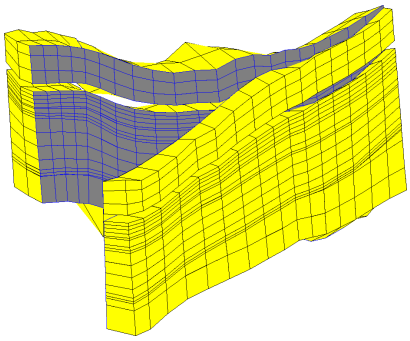
a) The whole model with active and inactive cells and four regions of interest marked in different colors



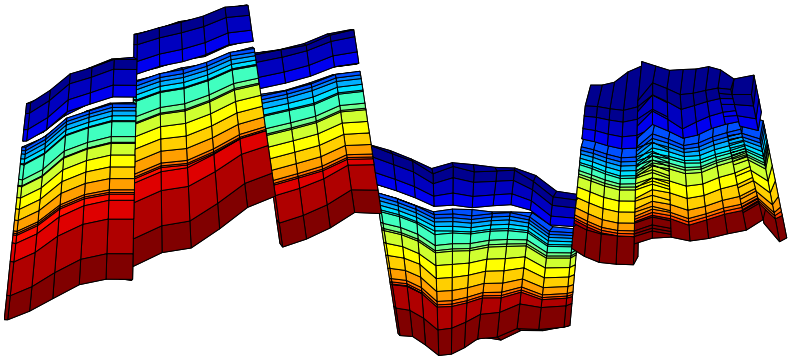
b) Zoom of the red region with pillars and corner-points shown as red circles



c) The magenta region with coloring according to cell volumes, which vary by a factor 700.



d) The blue region in which fault faces have been colored gray and the corresponding grid lines have been colored blue.



e) The green cross-section with coloring according to layer number from top to bottom of the model.

Fig. 3.19. Detailed view of subsets from the Norne simulation model.

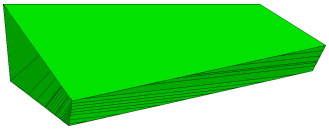
the letter Y. The pillars may also be piecewise polynomial curves, resulting in what is sometimes called S-faulted grids. Likewise, two neighboring pillars can collapse so that the basic grid shape becomes a prism rather than a hexahedron. However, there are several features that cannot easily be modelled, including multiple fault intersections (e.g., as in the letter 'F') and for this reason, the industry is constantly in search for improved gridding methods. One example is discussed in the next subsection. First, however, we will discuss some difficulties and challenges, seen from the side of a computational scientist seeking to use corner-point grids for computations.

The flexible cell geometry of the corner-point format poses several challenges for numerical implementations. Indeed, a geocellular grid is typically chosen by a geologist who tries to describe the rock body by as few volumetric cells as possible and who basically does not care too much about potential numerical difficulties his or her choice of geometries and topologies may cause in subsequent flow simulations.

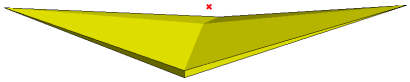
First of all, writing a robust grid-processing algorithm to compute geometry and topology or determine an equivalent matching, polyhedral grid can be quite a challenge. Displacements across faults will lead to geometrically complex, non-conforming grids, e.g., as illustrated in Figure 3.19. Since each face of a grid cell is specified by four (arbitrary) points, the cell interfaces in the grid will generally be bilinear, possibly strongly curved, surfaces. Geometrically, this can lead to several complications. Cell faces on different sides of a fault may intersect each other so that cells overlap volumetrically; cell faces need not be matching, which may leave void spaces; there may be tiny overlap ares between cell faces on different sides a fault; etc, which all contribute that fault geometries may be hard to interpret in a consistent way: a subdivision into triangles is, for instance, not unique. Likewise, top and bottom surfaces may intersect for highly curved cells with high aspect ratios, cell centroids may be outside the cell volume, etc.

The presence of degenerate cells, in which the corner-points collapse in pairs, implies that the cells will generally be polyhedral and possibly contain both triangular and bilinear faces (see Figure 3.13). Corner-point cells will typically be non-matching across faults or may have zero volume, which both introduces coupling between non-neighboring cells and gives rise to discretization matrices with complex sparsity patterns. All these facts call for flexible discretizations that are not sensitive to the geometry of each cell or the number of faces and corner points. Although not a problem for industry-standard two-point discretizations, it will pose implementational challenges for more advanced discretization methods that rely on the use of dual grids or reference elements. Figure 3.20 illustrates some geometrical and topological challenges seen in standard grid models.

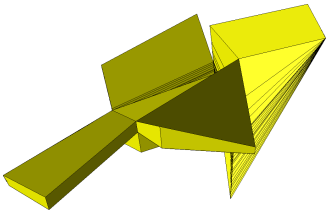
Third, to adapt to sloping faults, curved horizons and layers, lateral features, and so on, cell geometries may often deviate significantly from being orthogonal, which may generally introduce significant grid-orientation effects, in particular for the industry-standard two-point scheme (as we will see later).



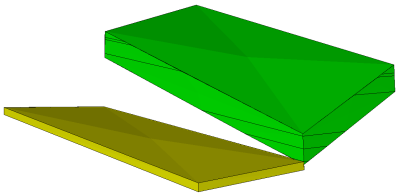
a) Many faces resulting from subdivision to give matching grid at faults



b) A $800 \times 800 \times 0.25$ m celll, very high aspect ratio and centroid outside cell



c) Difficult geometries



d) Small interface between two cells

Fig. 3.20. Illustration of difficulties and challenges associated with real-life corner-point geometries.

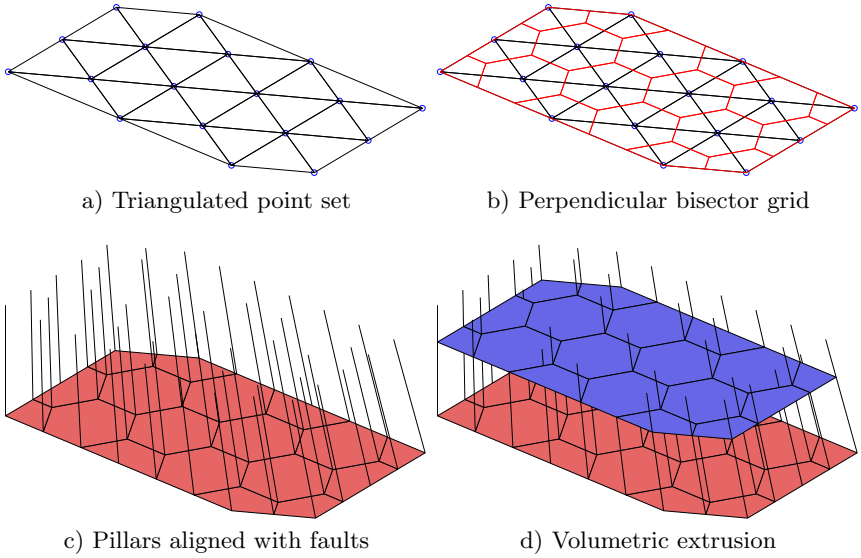
Fourth, stratigraphic grids will often have aspect ratios that are two or three orders of magnitude. Such high aspect ratios can introduce severe numerical difficulties because the majority of the flow in and out of a cell occurs across the faces with the smallest area. Similarly, the possible presence of strong heterogeneities and anisotropies in the permeability fields, e.g., as seen in the SPE 10 example in Chapter 2, typically introduces large condition numbers in the discretized flow equations.

Finally, corner-point grids generated by geological modeling typically contain too many cells. Once created by the geologist, the grid is handed to a reservoir engineer, whose first job is to reduce the number of cells if he or she is to have any hope of getting the model through a simulator. The generation of good coarse grids for use in upscaling, and the upscaling procedure itself, is generally work-intensive, error prone, and not always sufficiently robust, as we will come back to later in the book.

3.3.2 Layered 2.5D PEBI Grids

Corner-point grids are well suited to represent stratigraphic layers and faults which laterally coincide with one of the coordinate directions. Although the great flexibility inherent in the corner-point scheme can be used to adapt to areally skewed or curved faults, or other areal features, the resulting cell geometries will typically deviate far from being orthogonal, and hence introduce numerical problems in a subsequent flow simulation, as discussed above.

So-called 2.5D PEBI grids are often used to overcome the problem of areal adaption. These grids have been designed to combine the advantages of two different gridding methods: the (areal) flexibility of unstructured grids



a) Triangulated point set b) Perpendicular bisector grid

c) Pillars aligned with faults d) Volumetric extrusion

Fig. 3.21. Illustration of a typical process for generating 2.5D PEBI grids.

and the orthogonality of Cartesian grids. The PEBI grids are constructed in much of the same way as corner-point grids. One can, for instance, start with a point set, generate a lateral along one or more horizons, construct a lateral polyhedral grid by connecting the perpendicular bisectors of the triangle edges, one define a set of pillars that align with the major faults, and use these to extrude the areal grid cells to a volumetric grid, see Figure 3.21. The areal grids are typically constructed starting with a set of points, from which a Delaunay triangulation and then a perpendicular bisector grid is constructed. The resulting volumetric grid now is unstructured in the lateral direction, but has a layered structure in the vertical direction (and can thus be indexed using a [I,K] index pair). Because the grid is unstructured in the lateral direction, there is quite large freedom in choosing the size and shape of the grid cells, meaning that (almost) any number of complex features can be modelled, including adaption to curved faults, accurate resolution of near-well phenomena, etc.

As a first example of a 2.5D PEBI grid, we will consider a simple box geometry and generate a grid that can be seen as the dual of the tetrahedral tessellation shown to the right in Figure 3.8:

```
N=7; M=5; [x,y] = ndgrid(0:N,0:M);
x(2:N,2:M) = x(2:N,2:M) + 0.3*randn(N-1,M-1);
y(2:N,2:M) = y(2:N,2:M) + 0.3*randn(N-1,M-1);
aG = pebi(triangleGrid([x(:) y(:)]));
G = makeLayeredGrid(aG, 3);
plotGrid(G, 'FaceColor',[.8 .8 .8]);  view(-40,60); axis tight off
```

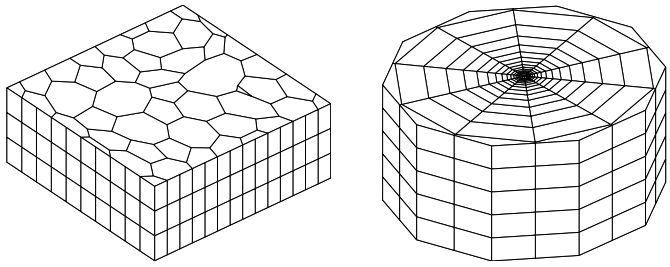


Fig. 3.22. The left plot shows a 2.5D Voronoi grid derived from a perturbed 2D point mesh extruded in the z-direction, whereas the right plot shows a radial grid.

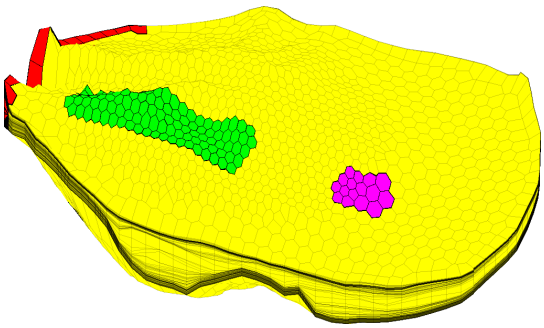
That is, we first construct a lateral 2D Voronoi grid from a set of generating points obtained by perturbing the vertices of a regular Cartesian grid, then use the function `makeLayeredGrid` to extrude this Voronoi grid to 3D along vertical pillars in the z-direction. The resulting grid is shown in the left plot of Figure 3.22.

As a second example, we will generate a radial grid that is graded towards the origin

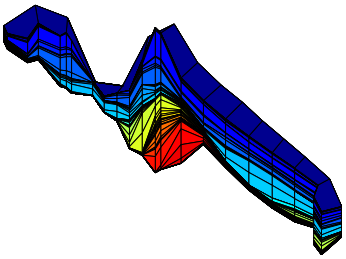
```
P = [];
for r = exp(-3.5:.25:0),
   [x,y,z] = cylinder(r,16); P = [P [x(1,:); y(1,:)]];
end
P = unique([P'; 0 0], 'rows');
G = makeLayeredGrid(pebi(triangleGrid(P)), 5);
plotGrid(G,'FaceColor',[.8 .8 .8]); view(30,50), axis tight off
```

giving the grid shown to the right in Figure 3.22. Typically, the main difficulty lies in generating a good point set (and a set of pillars). Once this is done, the rest of the process is almost straightforward.

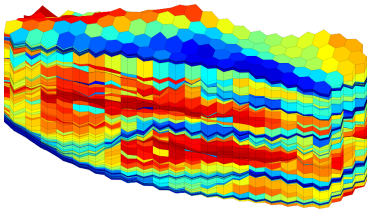
Our third example is a simulation model of a real reservoir. The model shown in Figure 3.23 consists of an unstructured areal grid that has been extruded vertically to model different geological layers. Some of the layers are very thin, which can be seen in particular in Figure 3.23a in which these thin layers appear as if they were thick lines. Figure 3.23b shows part of the perimeter of the model; we notice that the lower layers (yellow to red colors) have been eroded away in most of the grid columns, and although the vertical dimension is strongly exaggerated, we see that the layers contain steep slopes. To a non-geologist looking at the plot in Figure 3.23e, it may appear as if the reservoir was formed by sediments being deposited along a sloping valley that ends in a flat plain. Figures 3.23c and d show more details of the permeability field inside the model. The layering is particularly distinct in plot d, which is sampled from the flatter part of the model. The cells in plot c, on the other hand, show examples of pinch-outs. The layering provides a certain structure in the model, and it is therefore common to add a logical ik index, similar



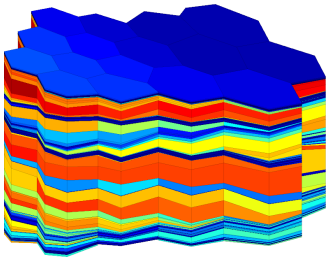
a) The whole model with three areas of interested marked in different colors.



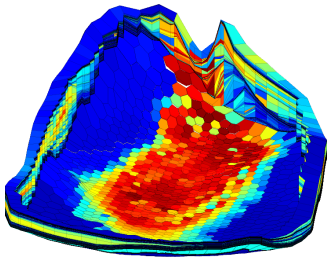
b) Layers 41 to 99 of the red region with colors representing the k-index.



c) Horizontal permeability in the green region of plot a).



d) Horizontal permeability in the magenta region of plot a).



e) Horizontal permeability along the perimeter and bottom of the model.

Fig. 3.23. A real petroleum reservoir modelled by a 2.5D PEBI grid having 1174 cells in the lateral direction and 150 cells along each pillar. Only 90644 out of the 176100 cells are active. The plots show the whole model as well as selected details.

to the logical ijk index for corner-point grids, where i refers to the areal numbering and k to the different layers. Moreover, it is common practice to associate a virtual logically Cartesian grid as an 'overlay' to the 2.5D grid that can be used e.g., to simplify lookup of cells in visualization. In this setup, more than one grid cell may be associated with a cell in the virtual grid.

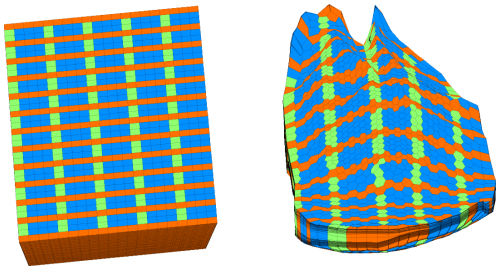


Fig. 3.24. Example of a virtual grid used for fast lookup in a 2.5D PEBI grid. The virtual grid has dimensions $37 \times 20 \times 150$ while the PEBI grid has ik dimensions 1174×150.

3.4 Grid Structure in MRST

In the two previous sections we have given an introduction to structured and unstructured grid types that can be created using MRST. In this section, we will go into more detail about the internal data structure used to represent various grid types. This data structure is in many ways the most fundamental part of MRST since almost all solvers and visualization routines require an instance of a grid as input argument. By convention, instances of the grid structure are denoted `G`. Readers who are mainly interested in *using* solvers and visualization routines already available in MRST, need no further knowledge of the grid structure beyond what has been encountered in the examples presented so far and can safely skip the remains of this section. For readers who wish to use MRST to prototype new computational methods, however, knowledge of the inner workings of the grid structure is essential. To read the MRST documentation, type

```
help grid_structure
```

As was stated in the introduction to the chapter, we have chosen to store all grid types using a general unstructured grid format that represents cells, faces, vertices, and connections between cells and faces. To this end, the main grid structure `G` contains three fields—`cells`, `faces`, and `nodes`—that specify individual properties for each individual cell/face/vertex in the grid. Grids in MRST can either be volumetric or lie on a 2D or 3D surface. The field `griddim` is used to distinguish volumetric and surface grids; all cells in a grid are polygonal surface patches if `griddim`=2 and polyhedral volumetric entities otherwise. In addition, the grid contains a field `type` consisting of a cell array of strings describing the history of grid-constructor and modifier functions through which a particular grid structure has been defined, e.g., 'tensorGrid'. For grids that have an underlying logical Cartesian structure, we also include the field `cartDims`.

The cell structure, `G.cells`, consists of the following mandatory fields:

– `num`: the number n_c of cells in the global grid.
– `facePos`: an indirection map of size [`num`+1,1] into the `faces` array. Specifically, the face information of cell `i` is found in the submatrix

`faces(facePos(i) : facePos(i+1)−1, :)`

The number of faces of each cell may be computed using the statement `diff(facePos)` and the total number of faces is given as n_f = `facePos(end)`−1.
– `faces`: an $n_f \times 3$ array that gives the global faces connected to a given cell. Specifically, if `faces(i,1)==j`, then global face number `faces(i,2)` is connected to global cell number j. The last component, `faces(i,3)`, is optional and can for certain types of grids contain a tag used to distinguish face directions: `East`, `West`, `South`, `North`, `Bottom`, `Top`.
The first column of `face` is redundant: it consists of each cell index j repeated `facePos(j+1)−facePos(j)` times and can therefore be reconstructed by decompressing a run-length encoding with the cell indices `1:num` as encoded vector and the number of faces per cell as repetition vector. Hence, to conserve memory, only the last two columns of `face` are stored, while the first column can be reconstructed using the statement:

```
rldecode(1:G.cells.num, diff(G.cells.facePos), 2) .'
```

This construction is used a lot throughout MRST and has therefore been implemented as a utility function inside `mrst−core/utils/gridtools`

```
f2cn = gridCellNo(G);
```

– `indexMap`: an optional $n_c \times 1$ array that maps internal cell indices to external cell indices. For models with no inactive cells, `indexMap` equals $1 : n_c$. For cases with inactive cells, `indexMap` contains the indices of the active cells sorted in ascending order. An example of such a grid is the ellipsoid in Figure 3.4 that was created using a fictitious domain method. For logically Cartesian grids, a map of cell numbers to logical indices can be constructed using the following statements in 2D:

```
[ij{1:2}] = ind2sub(dims, G.cells.indexMap(:));
ij        = [ij{:}];
```

and likewise in 3D:

```
[ijk{1:3}] = ind2sub(dims, G.cells.indexMap(:));
ijk        = [ijk{:}];
```

In the latter case, `ijk(i:)` is the global (I, J, K) index of cell i.

In addition, the cell structure can contain the following optional fields that typically will be added by a call to `computeGeometry`:

– `volumes`: an $n_c \times 1$ array of cell volumes
– `centroids`: an $n_c \times d$ array of cell centroids in $\mathbb{R}^d$

The face structure, `G.faces`, consists of the following mandatory fields:

- `num`: the number n_f of global faces in the grid.
- `nodePos`: an indirection map of size [`num`+1,1] into the `nodes` array. Specifically, the node information of face `i` is found in the submatrix
  `nodes(nodePos(i) : nodePos(i+1)−1, :)`
  The number of nodes of each face may be computed using the statement `diff(nodePos)`. Likewise, the total number of nodes is given as $n_n =$ `nodePos(end)`−1.
- `nodes`: an $N_n \times 2$ array of vertices in the grid. If `nodes(i,1)==j`, the local vertex i is part of global face number j and corresponds to global vertex `nodes(i,2)`. For each face the nodes are assumed to be oriented such that a right-hand rule determines the direction of the face normal. As for `cells.faces`, the first column of `nodes` is redundant and can be easily reconstructed. Hence, to conserve memory, only the last column is stored, while the first column can be constructed using the statement:

```
rldecode(1:G.faces.num, diff(G.faces.nodePos), 2) .'
```

- `neighbors`: an $n_f \times 2$ array of neighboring information. Global face i is shared by global cells `neighbors(i,1)` and `neighbors(i,2)`. One of the entries in `neighbors(i,:)`, but not both, can be zero, to indicate that face i is an external face that belongs to only one cell (the nonzero entry).

In addition to the mandatory fields, `G.faces` has optional fields that are typically added by a call to `computeGeometry` and contain geometry information:

- `areas`: an $n_f \times 1$ array of face areas.
- `normals`: an $n_f \times d$ array of **area weighted**, directed face normals in $\mathbb{R}^d$. The normal on face i points from cell `neighbors(i,1)` to cell `neighbors(i,2)`.
- `centroids`: an $n_f \times d$ array of face centroids in $\mathbb{R}^d$.

Moreover, `G.faces` can sometimes contain an $n_f \times 1$ (int8) array, `G.faces.tag`, that can contain user-defined face indicators, e.g., to specify that the face is part of a fault.

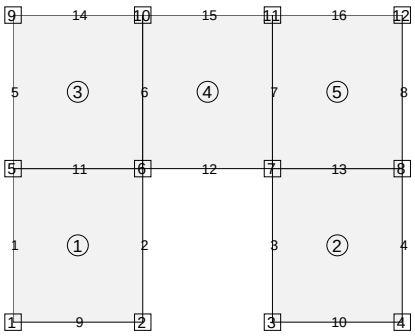
The vertex structure, `G.nodes`, consists of two fields:

- `num`: number N_n of global nodes (vertices) in the grid,
- `coords`: an $N_n \times d$ array of physical nodal coordinates in $\mathbb{R}^d$. Global node i is at physical coordinate `coords(i,:)`.

To illustrate how the grid structure works, we consider two examples. We start by considering a regular 3×2 grid, where we take away the second cell in the logical numbering,

```
G = removeCells( cartGrid([3,2]), 2)
```

This produces the output




```
cells.faces =          faces.nodes =      faces.neighbors =
 1   1   1 (East)          1   1              0   1
 1   9   3 (South)         1   5              1   0
 1   2   2 (West)          2   2              0   2
 1  11   4 (North)         2   6              2   0
 2   3   1 (East)          3   3              0   3
 2  10   3 (South)         3   7              3   4
 2   4   2 (West)          4   4              4   5
 2  13   4 (North)         4   8              5   0
 3   5   1 (East)          5   5              0   1
 3  11   3 (South)         5   9              0   2
 3   6   2 (West)          6   6              1   3
 3  14   4 (North)         6  10              0   4
 4   6   1 (East)          7   7              2   5
 4  12   3 (South)         7  11              3   0
 4   7   2 (West)          8   8              4   0
 4  15   4 (North)         8  12              5   0
 5   7   1 (East)          9   2
 5  13   3 (South)         9   1
 5   8   2 (West)          :   :
 5  16   4 (North)         :   :
```

Fig. 3.25. Illustration of the `cell` and `faces` fields of the grid structure: cell numbers are marked by circles, node numbers by squares, and face numbers have no marker.

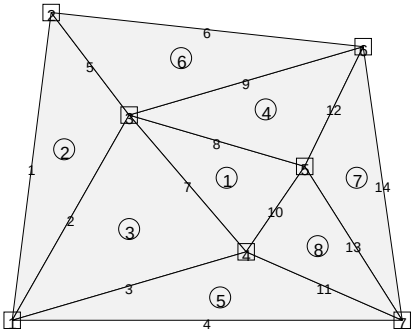
```
G =
        cells: [1x1 struct]
        faces: [1x1 struct]
        nodes: [1x1 struct]
    cartDims: [3 2]
        type: {'tensorGrid'  'cartGrid'  'removeCells'}
     griddim: 2
```

Examining the output from the call, we notice that the field `G.type` contains three values, 'cartGrid' indicates the creator of the grid, which again relies on 'tensorGrid', whereas the field 'removeCells' indicates that cells have been removed from the Cartesian topology. The resulting 2D geometry consists of five cells, twelve nodes, and sixteen faces. All cells have four faces and hence `G.cells.facePos` = [1 5 9 13 17 21]. Figure 3.25 shows[4] the geometry and topology of the grid, including the content of the fields `cells.faces`, `faces.nodes`, and `faces.neighbors`. We notice, in particular, that all interior faces (6, 7, 11, and 13) are represented twice in `cells.faces` as they belong to two different cells. Likewise, for all exterior faces, the corresponding row in `faces.neighbors` has one zero entry. Finally, being logically Cartesian, the grid structure contains a few optional fields:

- `G.cartDims` equals [3 2],
- `G.cells.indexMap` equals [1 3 4 5 6] since the second cell in the logical numbering has been removed from the model, and
- `G.cells.faces` contains a third column with tags that distinguish global directions for the individual faces.

As a second example, we consider an unstructured triangular grid given by seven points in 2D:

[4] To create the plot in Figure 3.25, we first called `plotGrid` to plot the grid, then called `computeGeometry` to compute cell and face centroids, which were used to place a marker and a text label with the cell/face number in the correct position.



```
cells.faces =
   1  10
   1   8
   1   7
   2   1
   2   2
   2   5
   3   3
   3   7
   3   2
   4   8
   4  12
   4   9
   5   3
   5   4
   5  11
   6   9
   6   6
   6   5
   7  13
   7  14
   :   :
```

```
faces.nodes =
   1   1
   1   2
   2   1
   2   3
   3   1
   3   4
   4   1
   4   7
   5   2
   5   3
   6   2
   6   6
   7   3
   7   4
   8   3
   8   5
   9   3
   9   6
  10   4
  10   5
   :   :
```

```
faces.neighbors =
   0   2
   2   3
   3   5
   5   0
   6   2
   0   6
   1   3
   4   1
   6   4
   1   8
   8   5
   4   7
   7   8
   0   7
```

Fig. 3.26. Illustration of the `cell` and `faces` fields of the grid structure: cell numbers are marked by circles, node numbers by squares, and face numbers have no marker. squares.

```
p = [ 0.0, 1.0, 0.9, 0.1, 0.6, 0.3, 0.75; ...
      0.0, 0.0, 0.8, 0.9, 0.2, 0.6, 0.45]'; p = sortrows(p);
G = triangleGrid(p)
```

which produces the output

```
G =
    faces: [1x1 struct]
    cells: [1x1 struct]
    nodes: [1x1 struct]
     type: {'triangleGrid'}
  griddim: 2
```

Because the grid contains no structured parts, `G` only consists of the three mandatory fields `cells`, `faces`, and `nodes` that are sufficient to determine the geometry and topology of the grid, the `type` tag naming its creator, and `griddim` giving that it is a surface grid. Altogether, the grid consists of eight cells, fourteen faces, and seven nodes, which are shown in Figure 3.26 along with the contents of the fields `cells.faces`, `faces.nodes`, and `faces.neighbors`. Notice, in particular, the absence of the third column in `cells.faces`, which generally does not make sense for a (fully) unstructured grid. Likewise, the `cells` structure does not contain any `indexMap` as all cells in the model are active.

Surface grids do not necessary have to follow a planar surface in 2D, but can generally be draped over a (continuous) surface in 3D. In MRST, such grids are used in the `co2lab` module for simulating CO_2 storage in deep saline aquifers using vertically-integrated models that describe the thickness of a supercritical CO_2 plume under a sealing caprock. To demonstrate the basic feature of a surface grid, we generate a 2D PEBI grid and drape it over MATLAB's `peaks` surface.

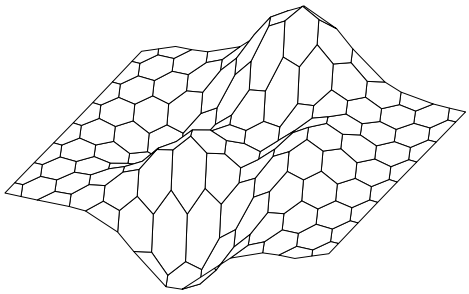


Fig. 3.27. Example of a surface grid: 2D PEBI grid draped over the `peaks` surface.

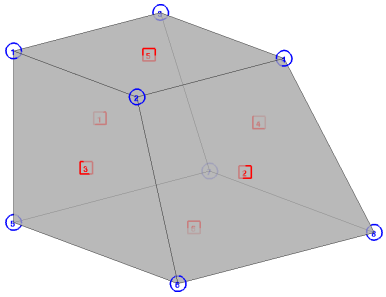
```
[x,y] = meshgrid([0:6]*2*cos(pi/6),0:7);
x = [x(:); x(:)+cos(pi/6)]; x=(x - mean(x(:)))/2;
y = [y(:); y(:)+sin(pi/6)]; y=(y - mean(y(:)))/2;
G = pebi(triangleGrid([x(:),y(:)]));
G.nodes.coords(:,3) = -peaks(G.nodes.coords(:,1),G.nodes.coords(:,2));
```

The resulting grid is shown in Figure 3.27. At the time of writing, the resulting grid is not a proper grid in the sense that although `computeGeometry` gives sensible results, the grid may not work properly with any of the flow and transport solvers supplied with MRST.

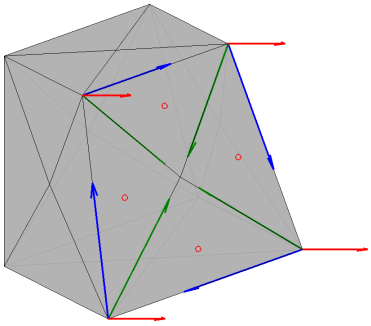
Computing geometry information

All grid factory routines in MRST generate the basic geometry and topology of a grid, that is, how nodes are connected to make up faces, how faces are connected to form cells, and how cells are connected over common faces. Whereas this information is sufficient for many purposes, more geometrical information may be required in many cases. As explained above, such information is provided by the routine `computeGeometry`, which computes cell centroids and volumes and face areas, centroids, and normals. Whereas computing this information is straightforward for simplexes and Cartesian grids, it is not so for general polyhedral grids that may contain curved polygonal faces. In the following we will therefore go through how it is done in MRST.

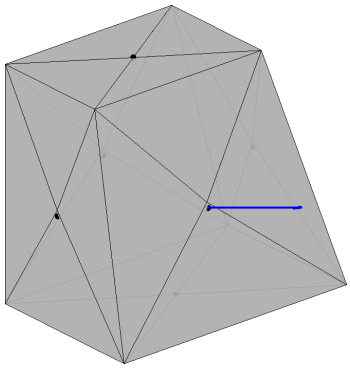
For each cell, the basic grid structure provides us with a list of vertices, a list of cell faces, etc, as shown in the upper-left plots of Figures 3.28 and 3.29. The routine starts by computing face quantities (areas, centroids, and normals). To utilize MATLAB efficiently, the computations are programmed using vectorization so that each derived quantity is computed for all points, all faces, and all cells in one go. To keep the current presentation as simple as possible, we will herein only give formulas for a single face and a single cell. Let us consider a single face given by the points $\vec{p}(i_1), \ldots, \vec{p}(i_m)$ and let $\alpha = (\alpha_1, \ldots, \alpha_m)$ denote a multi-index that describes how these points are



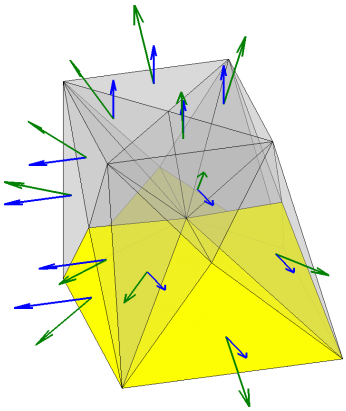
a) A single cell with face numbers (squares) and node numbers (circles)



b) Tessellation of faces with vectors $\vec{v}_1^k$ (blue), $\vec{v}_2^k$ (green), and $\vec{n}^k$ (red)



c) Face centroids and normal vectors computed from tessellation



d) Triangulation of cell volume with vectors $\vec{n}^k$ (blue) and $\vec{c}_r^k$ (green)

Fig. 3.28. Steps in the computation of geometry information for a single corner-point cell using `computeGeometry`.

connected to form the perimeter of the faces. For the face with global number j, the multi-index is given by the vector

`G.faces.nodes(G.faces.nodePos(j):G.faces.nodePos(j+1)-1)`

Let us consider two faces. Global face number two in Figure 3.28 is planar and consists of points $\vec{p}(2), \vec{p}(4), \vec{p}(6), \vec{p}(8)$ with the ordering $\alpha = (2, 4, 8, 6)$. Likewise, we consider global face number one in Figure 3.29, which is curved and consists of points $\vec{p}(1), \ldots, \vec{p}(5)$ with the ordering $\alpha = (4, 3, 2, 1, 5)$. For curved faces, we need to make a choice of how to interpret the surface spanned by the node points. In MRST (and some commercial simulators) this is done as follows: We start by defining a so-called *hinge point* $\vec{p}_h$, which is often given as part of the input specification of the grid. If not, we use the m points that make up the face and compute the hinge point as the centre point of

the face, $\vec{p}_h = \sum_{k=1}^{m} \vec{p}(\alpha_k)/m$. The hinge point can now be used to tessellate the face into m triangles, as shown to the upper left in Figures 3.28 and 3.29. The triangles are defined by the points $\vec{p}(\alpha_k)$, $\vec{p}(\alpha_{\mathrm{mod}(k,m)+1})$, and $\vec{p}_h$ for $k = 1, \ldots, m$. Each triangle has a center point $\vec{p}_c^{\,k}$ defined in the usual way as the average of its three vertexes and a normal vector and area given by

$$\vec{n}^k = \big(\vec{p}(\alpha_{\mathrm{mod}(k,m)+1}) - \vec{p}(\alpha_k)\big) \times \big(\vec{p}_h - \vec{p}(\alpha_k)\big) = \vec{v}_1^{\,k} \times \vec{v}_2^{\,k}$$
$$A^k = \sqrt{\vec{n}^k \cdot \vec{n}^k}.$$

The face area, centroid, and normal are now computed as follows

$$A_f = \sum_{k=1}^{m} A^k, \qquad \vec{c}_f = (A_f)^{-1} \sum_{k=1}^{m} \vec{p}_c^{\,k} A^k, \qquad \vec{n}_f = \sum_{k=1}^{m} \vec{n}^k. \tag{3.2}$$

The result is shown to the lower left in Figures 3.28, where the observant reader will see that the centroid $\vec{c}_f$ does not coincide with the hinge point $\vec{p}_h$ unless the planar face is a square. This effect is more pronounced for the curved faces of the PEBI cell in Figure 3.29.

The computation of centroids in (3.2) requires that the grid does not have faces with zero area, because otherwise the second formula would involve a division by zero and hence incur centroids with `NaN` values. The reader interested in creating his/her own grid-factory routines for grids that may contain degenerate (pinched) cells should be aware of this and make sure that all faces with zero area are removed in a preprocessing step.

To compute the cell centroid and volume, we start by computing the centre point $\vec{c}_c$ of the cell, which we define as the average of the face centroids, $\vec{c}_c = \sum_{k=1}^{m_f} \vec{c}_f/m_f$, where m_f is the number of faces of the cell. By connecting this centre point to the m_t face triangles, we define a unique triangulation of the cell volume, as shown to the lower right in Figures 3.28 and 3.29. For each tetrahedron, we define the vector $\vec{c}_r^{\,k} = \vec{p}_c^{\,k} - \vec{c}_c$, modify the triangle normals $\vec{n}^k$ so that they point outward, and compute the volume (which may be negative if the centre point $\vec{c}_c$ lies outside the cell)

$$V^k = \tfrac{1}{3} \vec{c}_r^{\,k} \cdot \vec{n}^k.$$

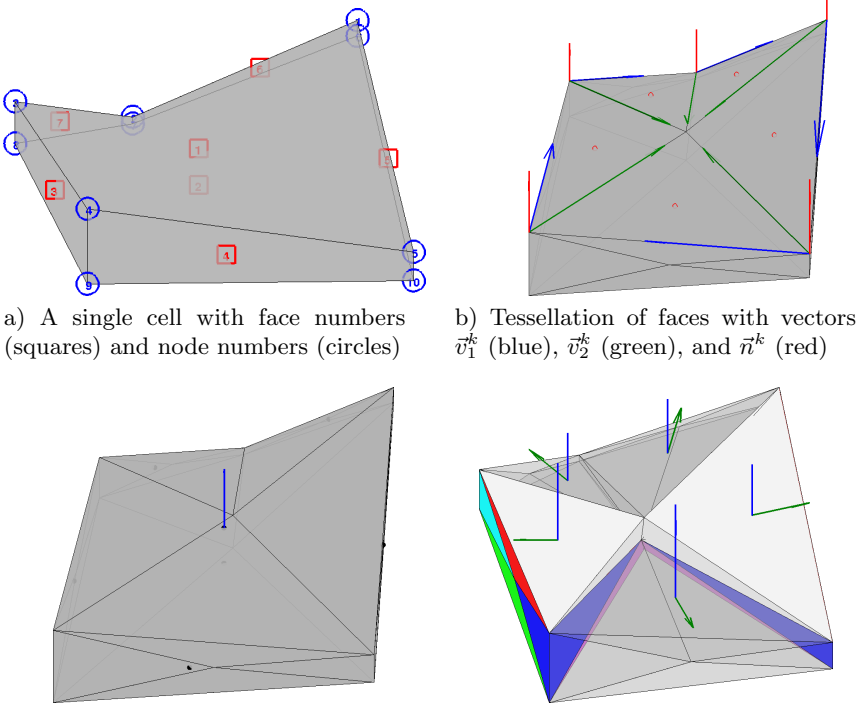
Finally, we can define the volume and the centroid of the cell as follows

$$V = \sum_{k=1}^{m_t} V^k, \qquad \vec{c} = \vec{c}_c + \frac{3}{4V} \sum_{k=1}^{m_t} V^k \vec{c}_r^{\,k}. \tag{3.3}$$

In MRST, all cell quantities are computed inside a loop, which may not be as efficient as the computation of the face quantities.

Example-grids in MRST

To help the user generate test cases, MRST supplies a routines for generating example grids. We have previously encountered `twister`, which perturbs the



a) A single cell with face numbers (squares) and node numbers (circles)

b) Tessellation of faces with vectors $\vec{v}_1^k$ (blue), $\vec{v}_2^k$ (green), and $\vec{n}^k$ (red)

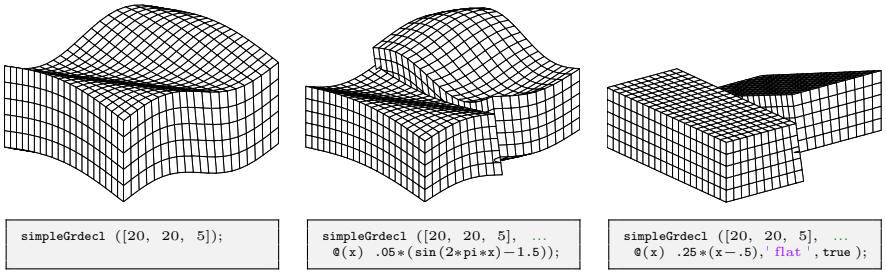
c) Face centroids and normal vectors computed from tessellation

d) Triangulation of cell volume with vectors $\vec{n}^k$ (blue) and $\vec{c}_r^k$ (green)

Fig. 3.29. Steps in the computation of geometry information for a single PEBI cell using `computeGeometry`.

x and y coordinates in a grid. Likewise, in Chapter 2.4 we used `simpleGrdecl` to generate a simple Eclipse input stream for a stratigraphic grid describing a wavy structure with a single deviated fault. The routine routine has several options that allow the user to specify the magnitude of the fault displacement, flat rather than a wavy top and bottom surfaces, and vertical rather than inclined pillars, see Figure 3.30.

Similarly, the routine with the somewhat cryptic name `makeModel3` generates a corner-point input stream that models parts of a dome that is cut through by two faults, see Figure 3.31. Similarly, `extrudedTriangleGrid.m` generates a 2.5D prismatic grid with a laterally curved fault in the middle. Alternatively, the routine can generate a 2.5D PEBI grid in which the curved fault is laterally stair-stepped, see Figure 3.31.

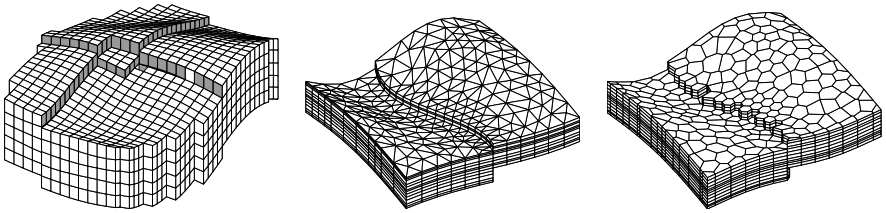


```
simpleGrdecl ([20, 20, 5]);
```

```
simpleGrdecl ([20, 20, 5], ...
  @(x) .05*(sin(2*pi*x)-1.5));
```

```
simpleGrdecl ([20, 20, 5], ...
  @(x) .25*(x-.5),'flat', true );
```

Fig. 3.30. The `simpleGrdecl` routine can be used to produce faulted, two-block grids of different shapes.



```
makeModel3 ([30,20,5]);
```

```
extrudedTriangleGrid (50);
```

```
extrudedTriangleGrid (50, true );
```

Fig. 3.31. Three different example grids created by the grid example functions `makeModel3` and `extrudedTriangleGrid`.

The SAIGUP Model

Having discussed the corner-point format in some detail, it is now time to return to the SAIGUP model. In the following, we will look at the grid representation in more detail and show some examples of how to interact and visualize different features of the grid. In Chapter 2.4, we saw that parsing the input file creates the following structure

```
grdecl =
    cartDims: [40 120 20]
       COORD: [29766x1 double]
       ZCORN: [768000x1 double]
      ACTNUM: [96000x1 int32]
       PERMX: [96000x1 double]
         :    :     :
```

In the following, we will (mostly) use the first four fields:

1. The dimension of the underlying logical Cartesian grid: Eclipse keyword `SPECGRID`, equal $40 \times 120 \times 20$.
2. The coordinates of the pillars: Eclipse keyword `COORD`, top and bottom coordinate per vertex in the logical 40×120 areal grid, i.e., $6 \times 41 \times 121$ values.

3. The coordinates along the pillars: Eclipse keyword `ZCORN`, eight values per cell, i.e., $8 \times 40 \times 120 \times 20$ values.
4. The boolean indicator for active cells: Eclipse keyword `ACTNUM`, one value per cell, i.e., $40 \times 120 \times 20$ values.

As we have seen above, we can use the routine `processGRDECL` to process the Eclipse input stream and turn the corner-point grid into MRST's unstructured description. The interested reader may ask the processing routine to display diagnostic output

```
G = processGRDECL(grdecl, 'Verbose', true);
G = computeGeometry(G)
```

and consult the SAIGUP tutorial (`saigupModelExample.m`) or the technical documentation of the processing routine for an explanation of the resulting output.

The model has been created using vertical pillars with lateral resolution of 75 meters and a vertical resolution of 4 meters, giving a typical aspect ratio of 18.75. This can be seen, e.g., by extracting the pillars and corner points and analyzing the results as follows:

```
[X,Y,Z] = buildCornerPtPillars(grdecl,'Scale',true);
dx = unique(diff(X)).'
[x,y,z] = buildCornerPtNodes(grdecl);
dz = unique(reshape(diff(z,1,3),1,[]))
```

The resulting grid has 78 720 cells that are almost equal in size (as can easily be seen by plotting `hist(G.cells.volumes)`), with cell volumes varying between 22 500 m^3 and 24 915 m^3. Altogether, the model has 264 305 faces: 181 649 vertical faces on the outer boundary and between lateral neighbors, and 82 656 lateral faces on the outer boundary and between vertical neighbors. Most of the vertical faces are not part of a fault and are therefore parallelograms with area equal 300 m^2. However, the remaining 26–27 000 faces are a result of the subdivision introduced to create a matching grid along the (stair-stepped) faults. Figure 3.32 shows where these faces appear in the model and a histogram of their areas: the smallest face has an area of $5.77{\cdot}10^{-4}$ m^2 and there are 43, 202, and 868 faces with areas smaller than 0.01, 0.1, and 1 m^2, respectively. The `processGRDECL` has an optional parameter `'Tolerance'` that sets the minimum distance used to distinguish points along the pillars (the default value is zero). By setting this to parameter to 5, 10, 25, or 50 cm, the area of the smallest face is increased to 0.032, 0.027, 0.097, or 0.604 m^2, respectively. In general, we advice against aggressive use of this tolerance parameter; one should instead develop robust discretization schemes and, if necessary, suitable post-processing methods that eliminate or ignore faces with small areas.

Next, we will show a few examples of visualizations of the grid model that will highlight various mechanisms for interacting with the grid and accessing parts of it. As a first example, we start by plotting the layered structure of

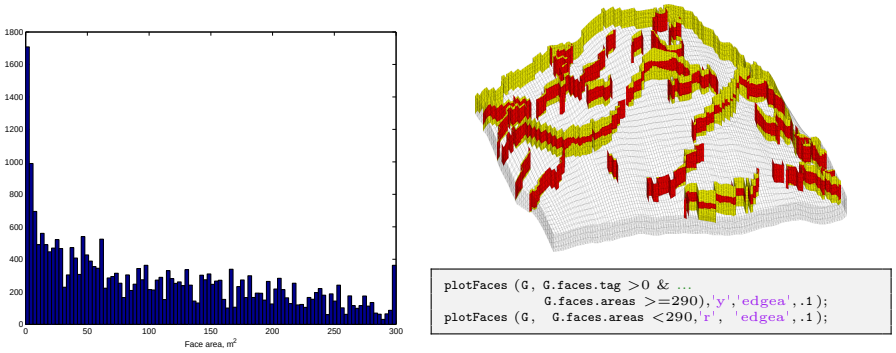


Fig. 3.32. Faces that have been subdivided for the SAIGUP mode. The left plot shows a histogram of the faces areas. The right plot shows all fault faces (yellow) and fault faces having area less than 290 m^2 (red).

the model. To this end, we use a simple trick: create a matrix with ones in all cells of the logical Cartesian grid and then do a cumulative summation in the vertical direction to get increasing values,

```
val = cumsum(ones(G.cartDims),3);
```

which we then plot using a standard call to `plotCellData`, see the left plot in Figure 3.33. Unfortunately, our attempt at visualizing the layered structure was not very successful. We therefore try to extract and visualize only the cells that are adjacent to a fault:

```
cellList = G.faces.neighbors(G.faces.tag>0, :);
cells    = unique(cellList(cellList>0));
```

In the first statement, we go through all faces and extract the neighbors of all faces that are marked with a tag (i.e., lies at a fault face). The list may have repeated entries if a cell is attached to more than one fault face and contain zeros if a fault face is part of the outer boundary. We get rid of these in the second statement, and can then plot the result using `plotCellData(G,val(G.cells.indexMap),cells)`, giving the result in the right plot of Figure 3.33. Let us inspect the fault structure in the lower-right corner of the plot. If we disregard using `cutGrdecl` as discussed on page 63, there are basically two ways we can extract parts of the model, that both rely on the construction of a map of cell numbers of logical indices. In the first method, we first construct a logical set for the cells in a logically Cartesian bounding box and then use the builtin function `ismember` to extract the members of `cells` that lie within this bounding box:

```
[ijk{1:3}] = ind2sub(G.cartDims, G.cells.indexMap); ijk = [ijk{:}];
[I,J,K] = meshgrid(1:9,1:30,1:20);
bndBox = find(ismember(ijk,[I(:), J(:), K (:)], 'rows'));
inspect = cells(ismember(cells,bndBox));
```

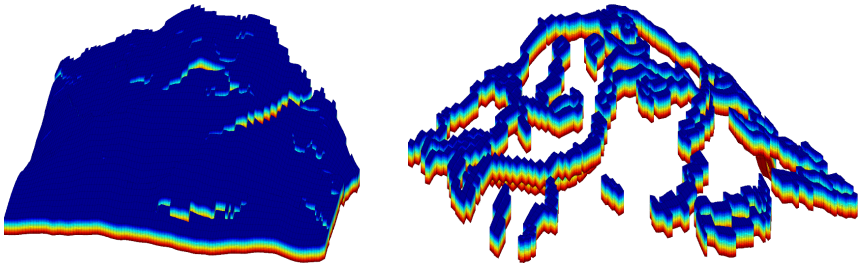


Fig. 3.33. Visualizing the layered structure of the SAIGUP model.

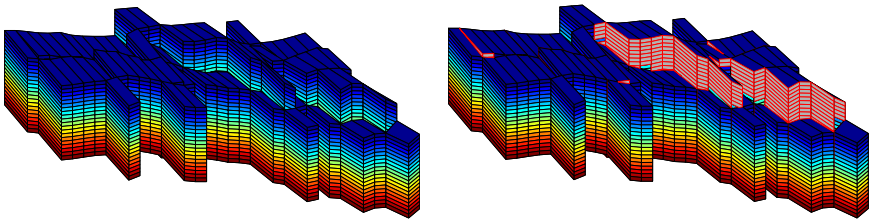


Fig. 3.34. Details from the SAIGUP model showing a zoom of the fault structure in the lower-right corner of the right plot in Figure 3.33. The left plot shows the cells attached to the fault faces, and in the right plot the fault faces have been marked with gray color and red edges.

The `ismember` function has an operational count of $\mathcal{O}(n \log n)$. A faster alternative is to use logical operations having an operational count of $\mathcal{O}(n)$. That is, we construct a vector of booleans that is `true` for the entries we want to extract and `false` for the remaining entries

```
[ijk{1:3}] = ind2sub(G.cartDims, G.cells.indexMap);

I = false(G.cartDims(1),1); I(1:9)=true;
J = false(G.cartDims(2),1); J(1:30)=true;
K = false(G.cartDims(3),1); K(1:20)=true;

pick  = I(ijk{1}) & J(ijk{2}) & K(ijk{3});
pick2 = false(G.cells.num,1); pick2(cells) = true;
inspect = find(pick & pick2);
```

Both approaches produce the same index set; the resulting plot is shown in Figure 3.34. To mark the fault faces in this subset of the model, we do the following steps

```
cellno  = rldecode(1:G.cells.num, diff(G.cells.facePos), 2) .';
faces   = unique(G.cells.faces(pick(cellno), 1));
inspect = faces(G.faces.tag(faces)>0);
plotFaces(G, inspect, [.7 .7 .7], 'EdgeColor','r');
```

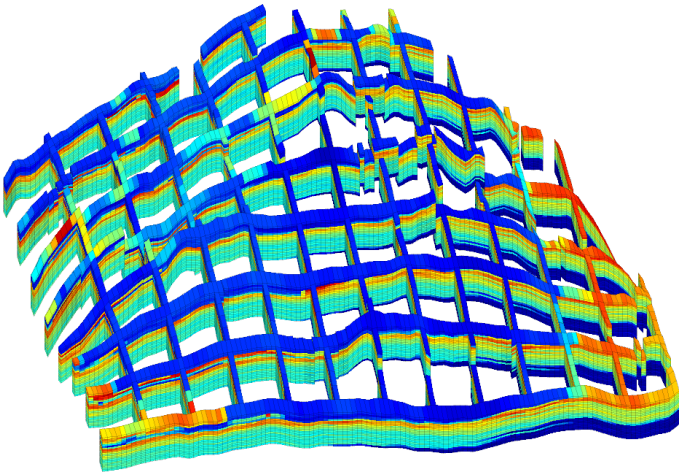


Fig. 3.35. A 'sieve' plot of the porosity in the SAIGUP model. Using this technique, one can more easily see the structure in the interior of the model.

The first statement constructs a list of all cells in the model, the second extracts a unique list of face numbers associated with the cells in the logical vector `pick` (which represents the bounding box in logical index space), and the third statement extracts the faces within this bounding box that are marked as fault faces.

Logical operations are also useful in other circumstances. As an example, we will extract a subset of cells forming a sieve that can be used to visualize the petrophysical quantities in the interior of the model:

```
% Every fifth cell in the x-direction
I = false(G.cartDims(1),1); I(1:5:end)=true;
J = true(G.cartDims(2),1);
K = true(G.cartDims(3),1);
pickX = I(ijk{1}) & J(ijk{2}) & K(ijk{3});

% Every tenth cell in the y-direction
I = true(G.cartDims(1),1);
J = false(G.cartDims(2),1); J(1:10:end) = true;
pickY = I(ijk{1}) & J(ijk{2}) & K(ijk{3});

% Combine the two picks
plotCellData(G,rock.poro, pickX | pickY, 'EdgeColor','k','EdgeAlpha',.1);
```

Composite Grids

One advantage of an unstructured grid description is that it easily allows the use of composite grids consisting of geometries and topologies that vary

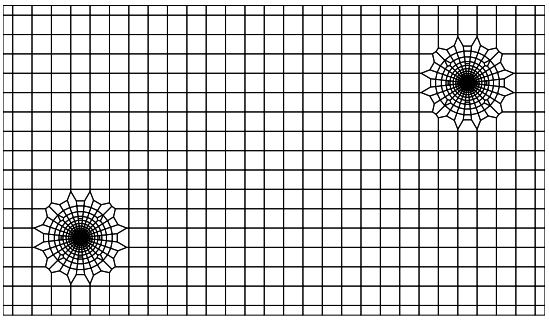


Fig. 3.36. A composite grid consisting of a regular Cartesian mesh with radial refinement around two well positions.

throughout the model. That is, different grid types or different grid resolution may be used locally to adapt to well trajectories and special features in the geology

As an example, we will generate a Cartesian grid that has a radial refinement around two wells in the interior of the domain. This composite grid will be constructed from a set of control points using the `pebi` routine. To this end, we first construct the generating point for a unit refinement, as discussed in Figure 3.22 above.

```
Pw = [];
for r = exp(-3.5:.2:0),
  [x,y,z] = cylinder(r,28); Pw = [Pw [x(1,:); y(1,:)]];
end
Pw = [Pw [0; 0]];
```

Then this point set is translated to the positions of the wells and glued into a standard regular point lattice (generated using `meshgrid`):

```
Pw1 = bsxfun(@plus, Pw, [2; 2]);
Pw2 = bsxfun(@plus, Pw, [12; 6]);
[x,y] = meshgrid(0:.5:14, 0:.5:8);
P = unique([Pw1'; Pw2'; x(:) y(:)], 'rows');
G = pebi(triangleGrid(P));
```

The resulting grid is shown in Figure 3.36. To get a good grid, it is important that the number of points around the cylinder has a reasonable match with the density of the points in the regular lattice. If not, the transition cells between the radial and the regular grid may exhibit quite unfeasible geometries. The observant reader will also notice the layer of small cells at the boundary, which is an effect of the particular distribution of the generating points (see the left plot in Figure 3.10) and can, if necessary be avoided by a more meticulous choice of points.

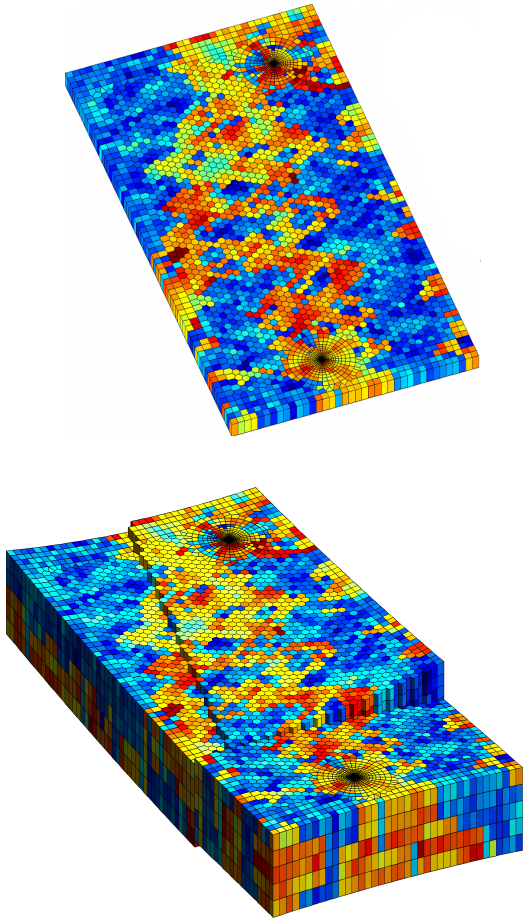


Fig. 3.37. Examples of composite grids. The upper plot shows an areal grid consisting of Cartesian, hexagonal, and radial parts. The lower plot shows the same grid extruded to 3D with two stair-stepped faults added.

More complex grids can be generated using the same approach. The left plot in Figure 3.37 shows an areal grid consisting of three parts: a Cartesian grid at the outer boundary, a hexagonal grid in the internal, and a radial grid with exponential radial refinement around two wells. The right plot shows an extruded 2.5D Voronoi grid that has been extruded to 3D along vertical pillars. In addition, structural displacement has been added along two areally stair-stepped faults that intersect near the west boundary. Petrophysical parameters have been sampled from layers 40–44 of the SPE10 data set [19].

4

Grid Coarsening

Over the last decades, methods for mapping and characterizing subsurface rock formations have improved tremendously. This, together with a dramatic increase in computational power, has enabled the industry to build increasingly detailed and complex models to account for heterogeneous structures on different spatial scales. Using the gridding techniques outlined above, one can today easily build complex geological models consisting of multiple million cells that account for most of the features seen in typical reservoirs. In fact, it is generally possible to identify many more geological layers and fine-scale heterogeneity features than it is practical to include in flow simulations.

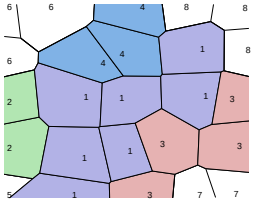
Through parallelization and use of high-performance computers it is possible to simulate fluid flow on grid models with multimillion cells, at least in principle. In many practical engineering workflows, however, such large-scale computations many not be feasible. The user may not have access to parallel hardware or software, or he/she may want to spend the available computational power on running a large number of model realizations instead of a few highly resolved ones. To obtain simulation models that are computationally tractable within a given computational budget, it is therefore common to develop reduced models through some kind of upscaling (homogenization) procedure that removes spatial detail from the geological description. Typically, a coarser model is developed by identifying regions consisting of several cells and then replacing each region by a single, coarse cell with homogeneous properties that represent the heterogeneity inside the region in some averaged sense. Upscaling, and alternative multiscale approaches, will be discussed later in the book.

Coarse grids required for upscaling can obviously be generated as discussed earlier in this chapter, using a coarser spatial resolution. However, this approach has two obvious disadvantages: First of all, if the original grid has a complex geometry, it will generally not be possible to preserve the exact geometry of the fine grid for an arbitrary coarse resolution. Second, there will generally not be a one-to-one mapping between cells in the fine and coarse grids. Herein, we have therefore chosen a different approach. *In MRST, a*

'coarse grid' always refers to a grid that is defined as a partition of another grid, which is referred to as the 'fine' grid. Tools for partitioning and coarsening of grids are found in two different modules of MRST: The `coarsegrid` module defines a basic grid structure for representing coarse grids, and supplies simple routines for partitioning logically Cartesian grids, whereas the `agglom` module offers tools for defining flexible coarse grids that adapt to geological features, flow patterns, etc. In the remains of this chapter, we will outline functionality for generating and representing coarse grids found in the `coarsegrid` module. In an attempt to distinguish fine and coarse grids, we will henceforth refer to fine grids as consisting of cells, whereas coarse grids are said to consist of blocks. (These two terms, 'cell' and 'block', are normally used interchangeable in the literature).

Coarse grids in MRST are represented by a structure that consists entirely of topological information stored in the same topological fields as for the general grid structure discussed in Section 3.4. As a naming convention, we will use `CG` to refer to a coarse-grid structure and `G` to refer to the usual (fine) grid. A coarse grid is always related to a fine grid in the sense that

- each cell in the fine grid `G` belongs to one, and only one, block in the coarse grid `CG`
- each block in `CG` consists of a *connected* subset of cells from `G`
- `CG` is defined by a partition vector p defined such that $p(i) = \ell$ if cell i in `G` belongs to block ℓ in `CG`



This concept is quite simple, but has proved to be very powerful in defining coarse grids that can be applied in a large variety of computational algorithms. We will come back to the details of the grid structure in Section 4.2. First, let us discuss how to define partition vectors in some detail as this is more useful from a user perspective than understanding the details of how the `CG` structure is implemented.

4.1 Partition Vectors

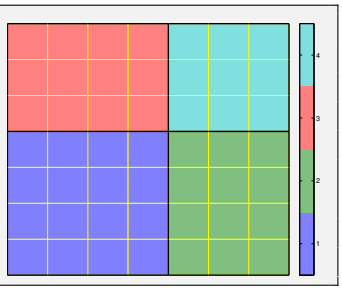
To demonstrate the simplicity and power of using a partition vectors to define coarse grids, we will go through a set of examples. Once you get the idea, it should be straightforward to use your own creativity for defining new partitions. (Complete codes for all examples discussed in this section are found in `showPartition.m`.)

4.1.1 Uniform Partitions

For all grids having a logically Cartesian structure, i.e., grids that have a valid field `G.cartDims`, one can use the function `partitionUI` to generate

a relatively uniform partition that will consist of the tensor product of a load-balanced linear partition in each index direction. As an example, let us partition a 7×7 fine grid into a 2×2 coarse grid:

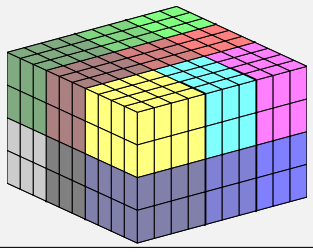
```
G = cartGrid([7,7]);
p = partitionUI(G, [2,2]);

plotCellData(G, p, 'EdgeColor', 'y');
outlineCoarseGrid(G, p, 'k');
axis tight off,
caxis([.5 max(p)+.5]);
colormap(lines(max(p)));
set(colorbar,'YTick',1:max(p));
```



The call to `partitionUI` returns a vector with one element per cell taking one of the integer values $1, 2, 3, 4$ that represent the four blocks. Since seven is not divisible by two, the coarse blocks will not have the same size: we will have blocks of size 4×4, 3×3, 4×3, and 3×4. To better distinguish the different blocks in the plot, we have used the function `outlineCoarseGrid(G, p)`, which will find and plot all faces in `G` for which the neighboring cells have different values of `p`.

The same procedure can, of course, also be applied to partition any grid in 3D that has a logically Cartesian topology as well. As an example, we consider a simple box geometry:

```
G = cartGrid([10,10,4]);
p = partitionUI(G, [3,3,2]);

plotCellData(G, p, 'Edgecolor', 'w');
outlineCoarseGrid(G, p, ...
      'EdgeColor','k','lineWidth',4);
colormap(colorcube(max(p)))
view(3); axis off
```

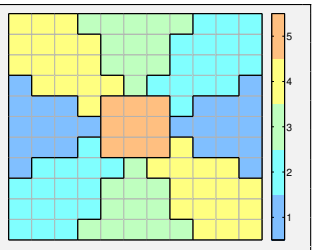


Here, we have used the `colorcube` colormap, which is particularly useful for visualizing partition vectors since it contains as many regularly spaced colors in RGB color space as possible. The careful reader will also observe that the arguments to `outlineCoarseGrid` changes somewhat for 3D grids.

4.1.2 Connected Partitions

All one generally needs to partition the grid is a partition vector, which can be given by the user, read from a file, generated by evaluating a function, or as the output of some user-specified algorithm. As a simple example of the latter, let us partition the box model $[-1, 1] \times [-1, 1]$ into nine different blocks using the polar coordinates of the cell centroids. The first block is defined as $r \leq 0.3$, while the remaining eight are defined by segmenting $4\theta/\pi$:

```
G = cartGrid([11, 11],[2,2]);
G.nodes.coords = ...
bsxfun(@minus, G.nodes.coords, 1);
G = computeGeometry(G);
c = G.cells.centroids;
[th,r] = cart2pol(c(:,1),c(:,2));
p = mod(round(th/pi*4)+4,4);
p(r<.3) = max(p)+1;
```



In the second last line, the purpose of the modulus operation is to avoid wrap-around effects as θ jumps from $-\pi$ to π.

While the human eye should be able to distinguish nine different coarse blocks in the plot above, the partition does unfortunately not satisfy all the criteria we prescribed on page 88. Indeed, as you can see from the colorbar, the partition vector only has five unique values and thus corresponds to five blocks according to our definition of `p`: cell i belongs to block ℓ if $p(i) = \ell$. Hence, what the partition describes is one connected block at the center surrounded by four disconnected blocks. A block is said to be disconnected if there are cells in the block that cannot be connected by a path that only crosses faces between cells inside the block. To get a grid that satisfies our requirements, we must split the four disconnected blocks. This can be done by the following call, which essentially forms an adjacency matrix for each block and finds the connected components by a Dulmage-Mendelsohn permutation (see `help dmperm`):

```
q = processPartition(G, p);
```

The routine will split cells that have the same `p`-value, but are not connected according to a face-neighborship topology into multiple blocks and update the partition vector accordingly. The routine can also take an additional parameter `facelist` that specifies a set of faces across which the connections will be removed before processing:

```
q = processPartition(G, p, facelist)
```

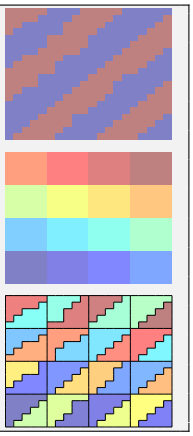
Using this functionality one can, for instance, prevent coarse blocks from crossing faults inside the model.

4.1.3 Composite Partitions

In many cases, the best way to create a good partition vector is by combining more than one partition criterion or principle. As an example, we can think of a heterogeneous medium consisting of two different facies, one with high permeability and one with low, that each form large contiguous regions. After coarsening, each coarse block will be assigned a homogeneous property, and it is therefore advantageous if the faces of the grid blocks follow the facies boundaries. Within each facies, we can use a standard Cartesian partition

generated by the routine `partitionCartGrid`, which is simpler and less computationally expensive than `partitionUI` but only works correctly if the fine grid has a fully intact logically Cartesian topology, i.e., if there are no inactive cells, no cells have been removed by `removeGrid`, and so on.

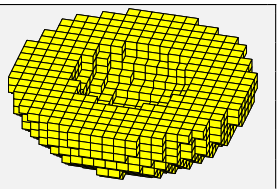
```
G = cartGrid([20, 20], [1  1]);
G = computeGeometry(G);

% Facies partition
f  = @(c) sin(4*pi*(c(:,1)-c(:,2)));
pf = 1 + (f(G.cells.centroids) > 0);

% Cartesian partition
pc = partitionCartGrid(G.cartDims, [4 4]);

% Alternative 1:
[b,i,p] = unique([pf, pc], 'rows' );
% Alternative 2:
q = compressPartition(pf + max(pf)*pc);
```



The example also shows two alternative techniques for combining different partitions. In the first alternative, we collect the partitions as columns in an array `A`. The call `[b,i,p]=unique(A,'rows')` will return `b` as the unique rows of `A` so that `b=A(i)` and `A=b(p)`. Hence, `p` will be a partition vector that represents the unique intersection of all the partitions collected in `A`. In the second method, we treat the partition vectors as multiple subscripts, from which we compute a linear index. This index may obviously not be contiguous. To avoid having to treat special cases that may arise from non-contiguous partition vectors, most routines in MRST require that partition vectors are contiguous. To fulfill this requirement, we use the function `compressPartition` that renumbers a partition vector to remove any indices corresponding to empty grid blocks. The two alternatives have more or less the same computational complexity, and which alternative you choose in your implementation is largely a matter of what you think will be easiest to understand for others who may have to go through your code.

Altogether, the examples above explain the basic concepts of how the authors tend to create partition vectors. However, before we go on to explain details of the coarse-grid structure and how to generate this structure from a given partition vector, we will show one last example that is a bit more fancy. To this end, we will create a cup-formed grid, partition it, and then visualize the partition using a nice technique. To generate the cup-shaped grid, we use the same fictitious-domain technique that we previously used for the ellipsoidal grid shown in Figure 3.4 on page 49.

```
x = linspace(−2,2,41);
G = tensorGrid(x,x,x);
G = computeGeometry(G);
c = G.cells.centroids;
r = c(:,1).^2 + c(:,2).^2+c(:,3).^2;
G = removeCells(G, (r>1) | (r<0.25) | (c(:,3)<0));
```



Assume that we wish to partition this cup model into one hundred coarse blocks. To this end, we could, for instance try to use `partitionUI` to impose a regular $5 \times 5 \times 4$ partition. Because of the fictitious method, a large number of the ijk indices from the underlying Cartesian topology will correspond to cells that are not present in the actual grid. Imposing a regular Cartesian partition on such a grid will typically give block indices in the range $[1, \max(p)]$ that do not correspond to any cells in the underlying fine grid. In this particular case, only seventy-nine out of the one desired one hundred blocks will correspond to a volume that is within the grid model. To see this, we used the function `accumarray` to count the number of cells for each block index and plotted the result as a bar chart:

```
subplot(2,1,1);
p = partitionUI(G,[5 5 4]);
bar(accumarray(p,1)); shading flat

q = compressPartition(p);
subplot(2,1,2);
bar(accumarray(q,1)); shading flat
set(gca,'XLim',[0 100]);
```

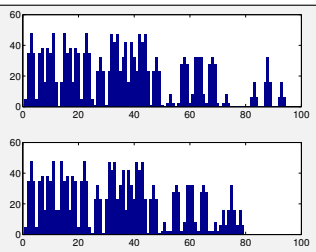


Figure 4.1 shows the partition obtained after we have compressed the partition vector. To clearly distinguish the different blocks, we have used an explosion view, which is a useful technique for visualizing coarse partitions. The complete code for this visualization method can be found in `explosionView.m`.

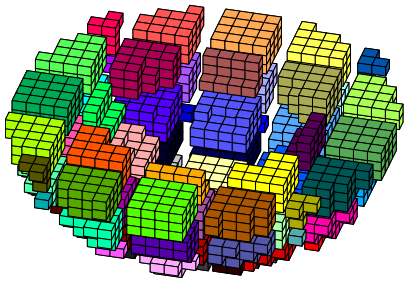


Fig. 4.1. The partition of the cup-formed grid visualized using an 'explosion view' type method.

4.2 Coarse Grid Representation in MRST

Now that you have seen how to generate coarse partitions, it is time to introduce how partition vectors can be developed into coarse grids. Given a grid structure `G` and a partition vector `p`, the structure `CG` representing the coarse grid can be generated by the following call:

```
CG = generateCoarseGrid(G, p)
```

The coarse-grid structure `CG` consists entirely of topological information stored in the same way as described in Section 3.4 for `G`: the fields `cells` and `faces` represent the coarse blocks and their connections. As a result, `CG` can be used seamlessly with many of the standard (incompressible) solvers in MRST. Unlike the original grid structure, however, `CG` does not represent the geometry of the coarse blocks and faces explicitly and does therefore not have a `nodes` field. The geometry information can instead be obtained from the parent grid `G` and the partition vector `p`; copies of these are stored within the coarse grid in the fields `parent` and `partition`, respectively.

The structure, `CG.cells`, that represents the coarse blocks consists of the following mandatory fields:

– `num`: the number N_b of blocks in the coarse grid.
– `facePos`: an indirection map of size [`num`+1,1] into the `faces` array, which is defined completely analogously as for the fine grid. Specifically, the connections information of block `i` is found in the submatrix
`faces(facePos(i) : facePos(i+1)−1, :)`
The number of connections of each block may be computed using the statement `diff(facePos)`.
– `faces`: an $N_c \times 2$ array of connections associated with a given block. Specifically, if `faces(i,1)==j`, then connection `faces(i,2)` is associated with block number j. To conserve memory, only the second column is actually stored in the grid structure, and can be reconstructed by a call to `rldecode`. Optionally, one may append a third column that contains a tag that has been inherited from the parent grid.

In addition, the cell structure can contain the following optional fields that typically will be added by a call to `CG=coarsenGeometry(CG)`, assuming that the corresponding information is available in the parent grid:

– `volumes`: an $N_b \times 1$ array of block volumes
– `centroids`: an $N_b \times d$ array of block centroids in $\mathbb{R}^d$

The face structure, `CG.faces`, consists of the following mandatory fields:

– `num`: the number N_c of global connections in the grid.
– `neighbors`: an $N_c \times 2$ array of neighboring information. Connection i is between blocks `neighbors(i,1)` and `neighbors(i,2)`. One of the entries in `neighbors(i,:)`, but not both, can be zero, to indicate that connection i is between a single block (the nonzero entry) and the exterior of the grid.

– `connPos`, `fconn`: packed data-array representation of the coarse → fine mapping. Specifically, the elements `fconn(connPos(i):connPos(i+1)−1)` are the connections in the parent grid (i.e., rows in `G.faces.neighbors`) that constitute coarse-grid connection `i`.

In addition to the mandatory fields, `CG.faces` has optional fields that are typically added by a call to `coarsenGeometry` and contain geometry information:

– `areas`: an $N_c \times 1$ array of face areas.
– `normals`: an $N_c \times d$ array of accumulated area-weighted, directed face normals in $\mathbb{R}^d$.
– `centroids`: an $N_c \times d$ array of face centroids in $\mathbb{R}^d$.

Like in `G`, the coarse grid structure also contains a field `CG.griddim` that is used to distinguish volumetric and surface grids, as well as a cell array `CG.type` of strings describing the history of grid-constructor and modifier functions used to define the coarse grid.

As an illustrative example, let us partition a 4×4 Cartesian grid into a 2×2 coarse grid. This gives the following structure:

```
CG =
        cells: [1x1 struct]
        faces: [1x1 struct]
    partition: [16x1 double]
       parent: [1x1 struct]
      griddim: 2
         type: {'generateCoarseGrid'}
```

with the `cells` and `faces` fields given as

```
CG.cells =
        num: 4
    facePos: [5x1 double]
      faces: [16x2 double]
```

```
CG.faces =
          num: 12
    neighbors: [12x2 double]
      connPos: [13x1 double]
        fconn: [24x1 double]
```

Figure 4.2 shows relations between entities in the coarse grid and in its parent grid. For instance, we see that block number one consists of cells one, two, five and six because these are the rows in `CG.partition` that have value equal one. Likewise, we see that because `CG.faces.connPos`(1:2)=[1 3], coarse connection number one is made up of two cell faces that correspond to faces number one and six in the parent grid because `CG.faces.fconn`(1:2)=[1 6], and so on.

4.2.1 Subdivision of Coarse Faces

In the discussion above, we have always assumed that there is only a single connection between two neighboring coarse blocks and that this connection is built up of a set of cell faces corresponding to all faces between pairs of cells in the fine grid that belong to the two different blocks. While this definition is useful for many workflows like in standard upscaling methods, there are

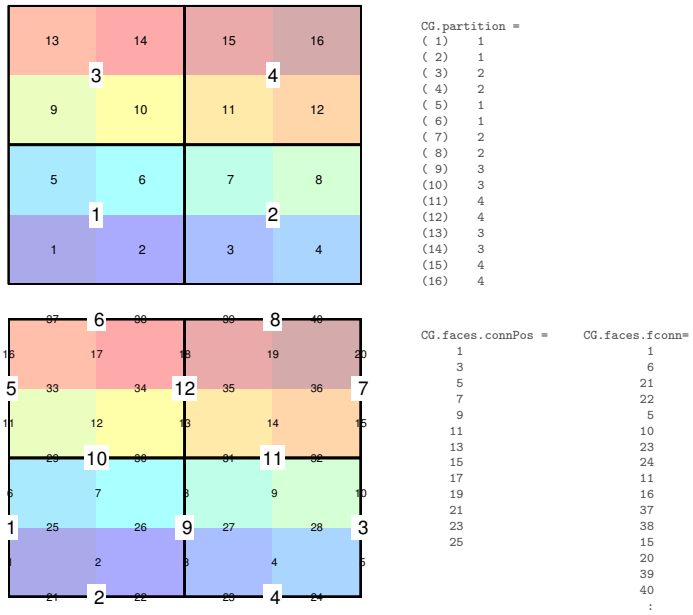


Fig. 4.2. Illustration of the relation between blocks in the coarse grid and cells in the parent grid (top) and between connections in the coarse grid and faces from the parent grid (bottom).

also problems for which one may want to introduce more than one connection between neighboring blocks. To define the subdivision of the coarse faces, we will once again use a partition vector with one scalar value per face in the fine grid, i.e., defined completely analogous to the partition vectors used for the volumetric partition. Assuming that we have two such partition vectors, `pv` describing the *volumetric* partition and `pf` describing the partition of cell faces, the corresponding coarse grid is built through the call:

```
CG = generateCoarseGrid(G, pv, pf);
```

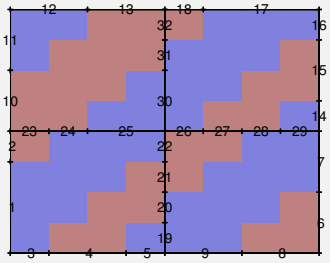
In our experience, the simplest way to build a face partition is to compute it from an ancillary volumetric partition using the routine:

```
pf = cellPartitionToFacePartition(G, pv)
```

which assigns a unique, non-negative integer for each pair of cell values occurring in the volumetric partition vector `pv`, and hence constructs a partitioning of all faces in the grid. Fine-scale faces that are not on the interface between coarse blocks are assigned zero value.

As an illustration, we continue the example from page 91 and use facies information to subdivide coarse faces so that each connection corresponds to a given combination of facies values on opposite sides of the interface:

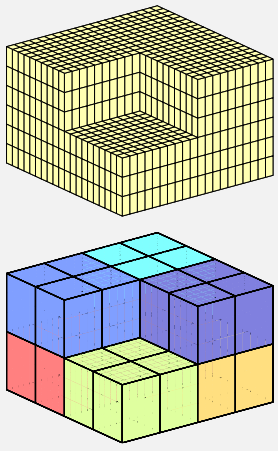
```
G  = computeGeometry(cartGrid([8, 8], [1 1]));
f  = @(c) sin(3*pi*(c(:,1)-c(:,2)));
pf = 1 + (f(G.cells.centroids) > 0);
plotCellData(G, pf); axis off;
colormap((jet(16)+ones(16,3))/2);

pv = partitionCartGrid(G.cartDims, [2 2]);
pf = cellPartitionToFacePartition(G,pf);
pf = processFacePartition(G, pv, pf);
CG = generateCoarseGrid(G, pv, pf);

CG = coarsenGeometry(CG);
plotFacesNew(CG,1:CG.faces.num,'Marker','+','MarkerSize',8,'LineWidth',2);
text(CG.faces.centroids(:,1), CG.faces.centroids(:,2), ...
  num2str((1:CG.faces.num)'),'FontSize',20, 'HorizontalAlignment','center');
```



As for the volumetric partition, we require that each interface that defines a connection in the face partition consists of a connected set of cell faces. That is, it must be possible to connect any two cell faces belonging to given interface by a path that only crosses edges between cell faces that are part of the interface. To ensure that all coarse interfaces are connected collections of fine faces, we have used the routine `processFacePartition`, which splits disconnected interfaces into one or more connected interfaces.

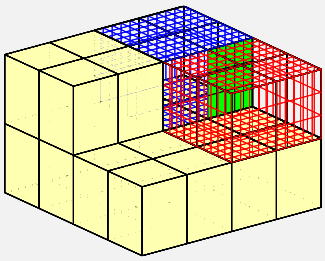
The same principles apply equally well in three dimensions, as shown in the next example:

```
G = computeGeometry(cartGrid([20 20 6]));
c = G.cells.centroids;
G = removeCells(G, ...
    (c(:,1)<10) & (c(:,2)<10) & (c(:,3)<3));
plotGrid(G); view(3); axis off

p = partitionUI(G,[2, 2, 2]);
q = partitionUI(G,[4, 4, 2]);
CG = generateCoarseGrid(G, p, ...
    cellPartitionToFacePartition(G,q));

plotCellDataNew(CG,(1:max(p))');
plotFacesNew(CG,1:CG.faces.num,...
    'FaceColor' , 'none' , 'LineWidth' ,2);
view(3); axis off
```



The coarse-grid structure `CG` also contains lookup tables for mapping blocks and interfaces in the coarse grid to cells and faces in the fine grid. To demonstrate this, we will visualize one connection of a subdivided coarse face that consists of several fine faces, along with the fine cells that belong to the neighboring blocks:

```
face  = 66;
sub   = CG.faces.connPos(face):CG.faces.connPos(face+1)-1;
ff    = CG.faces.fconn(sub);
neigh = CG.faces.neighbors(face,:);

show = false(1,CG.faces.num);
show(boundaryFaces(CG)) = true;
show(boundaryFaces(CG,neigh)) = false;
plotFacesNew(CG, show,'FaceColor',[1 1 .7]);
plotFacesNew(G, ff, 'FaceColor', 'g')
plotFacesNew(CG,boundaryFaces(CG,neigh), ...
   'FaceColor','none','LineWidth', 2);
plotGrid(G, p == neigh(1), 'FaceColor', 'none', 'EdgeColor', 'r')
plotGrid(G, p == neigh(2), 'FaceColor', 'none', 'EdgeColor', 'b')
```



4.3 Coarsening of Realistic Reservoir Models

To demonstrate that the coarsening principles outlined above can be applied to realistic models, we finish the chapter by discussing how to coarsen two corner-point models of industry-standard complexity: the sector model of the Johansen aquifer introduced in Section 2.4.4 and the SAIGUP model from Section 2.4.5. The complete code for the two examples can be found in the scripts `coarsenJohansen.m` and `coarsenSAIGUP.m`.

4.3.1 The Johansen Aquifer

As we saw in Section 2.4.4, the heterogeneous 'NPD5' sector model of the Johansen aquifer developed as a collaboration between the Norwegian Petroleum Directorate and researchers from the University of Bergen contains three different formations: the Johansen sandstone delta bounded above by the Dunlin shale and below by the Amundsen shale. As shown in Figure 2.11, these three formations have distinctively different permeabilities. The model was originally developed to study a potential site for geological storage of CO_2 injected as a supercritical fluid deep in the formation. The injected CO_2 is much lighter than the resident brine and will form a separate, buoyant phase. The low-permeable Dunlin shale will act as a caprock, under which the CO_2 will accumulate as a thin plume that migrates upward under the caprock in the up-dip direction. The formations play a very different role in the sequestration process, the Johansen sandstone is the container in which the CO_2 is to be kept, while the Dunlin shale acts like a seal that prevents the CO_2 to escape back to the sea bottom. To accurately simulate the sequestration process, it is therefore important to preserve the three formations in the coarse model and avoid creating coarse blocks that contain large media contrasts. We will

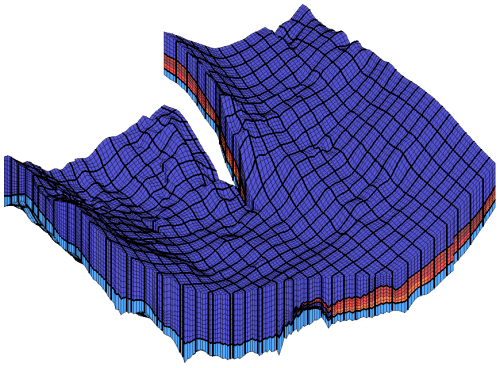


Fig. 4.3. A $4 \times 4 \times 11$ coarsening of the NPD5 model of the Johansen aquifer that preserves the Amundsen, Dunlin, and Johansen formations.

Table 4.1. Permeability values used to distinguish the different formations in the NPD5 sector model of the Johansen formation.

Dunlin	Johansen	Amundsen
$\mathtt{K} \leq 0.01$mD	0.1 mD $< \mathtt{K}$	0.01 mD $< \mathtt{K} \leq 0.1$mD

therefore coarsen the three formations separately. To distinguish which formation each particular cell belongs to, we will use permeability values `K` as indicator as shown in Table 4.1

Likewise, to correctly resolve the formation and migration of the thin plume it is essential that the grid has as high vertical resolution as possible. In a real simulation, we would therefore normally only reduce the lateral resolution, say by a factor four in each lateral direction. Here, however, we first use only a single block in the vertical direction inside each formation to more clearly demonstrate how the coarsening can adapt to the individual formations.

Assuming that the grid `G` and the permeability field `K` in units [mD] have been initialized properly as described in Section 2.4.4, the coarsening procedure reads

```
pK = 2*ones(size(K)); pK(K<=0.1) = 3; pK(K<=0.01)= 1;
pC = partitionUI(G, [G.cartDims(1:2)/4 1]);
[b,i,p] = unique([pK, pC], 'rows');
p = processPartition(G,p);
CG = generateCoarseGrid(G, p);
plotCellData(G,log10(K),'EdgeColor','k','EdgeAlpha',.4); view(3)
outlineCoarseGrid(G,p,'FaceColor','none','EdgeColor','k','LineWidth',1.5);
```

The resulting coarse grid is shown in Figure 4.3. By intersecting the partition vector `pC`, which has only one block in the vertical direction, with the partition vector `pK` that represents the different formations, we get three blocks in

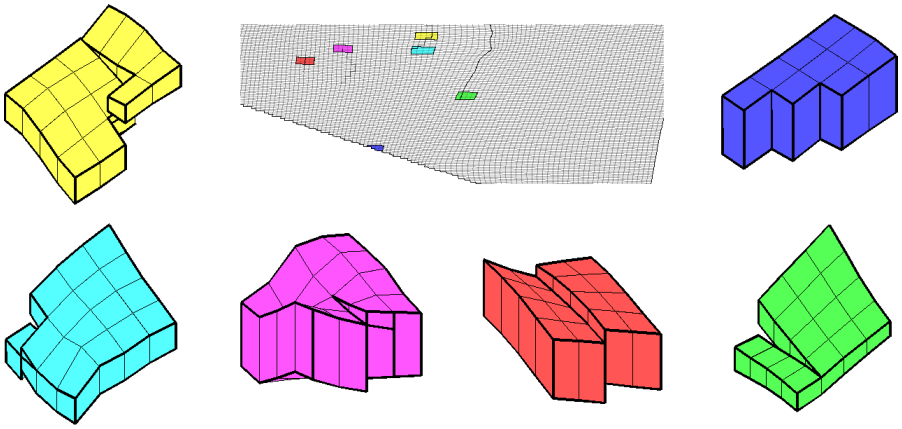


Fig. 4.4. Six coarse blocks sampled from the top grid layer of the Dunlin formation in a $4 \times 4 \times 1$ coarsening of the NPD5 sector model of the Johansen formation.

the vertical direction where all formations are present and two blocks in the vertical direction where only the Dunlin and Amundsen shales are present.

The aquifer model contains one big and several small faults. As a result, 1.35% of the cells in the original grid have more than six neighbors. Coarsening a model that has irregular geometry (irregular perimeter, faults, degenerate cells, etc) uniformly in index space will in most cases give quite a few blocks whose geometry deviate quite a lot from being rectangular, and the resulting coarse grid will generally contain a larger percentage of unstructured connections than the original fine model. For the NPD5 aquifer model, 20.3% of the blocks in the coarse model have more then six coarse faces. If we look at a more realistic coarsening that retains the vertical resolution of the original model, 16.5% of the blocks have more than six neighboring connections. This model is obtained if we repeat the construction above using

```
pC = partitionUI(G, G.cartDims./[4 4 1]);
```

Figure 4.4 six different coarse blocks sampled from the top grid layer of the Dunlin shale. Here, the blue block has an irregular geometry because it is sampled from a part of the aquifer perimeter that does not follow the primary grid directions. The other five blocks contain (parts of) a fault and will therefore potentially have extra connections to blocks in grid layers below. Despite the irregular geometry of the blocks, the coarse grid can be used directly with the basic incompressible flow and transport solvers described in Chapter 6. In our experience, the quality of the coarse solution will generally be more affected by the quality of the upscaling of the petrophysical parameters (see Chapter **??**) than the irregularity of the block geometry. In fact, having irregular blocks that preserve geometry of the fine-scale model will respect the layering and connections in the fine-scale geology and therefore often give more accurate results than a coarse model with hexahedral blocks.

4.3.2 The Shallow-Marine SAIGUP Model

As our next example, we will revisit the SAIGUP model discussed on page 79. As shown in Figure 2.16 in Section 2.4.5, the model has six user-defined rock types (also known as saturation regions) that can be used to specify different rock-fluid behavior. Depending upon the purpose of the reduced model, one may want to preserve these rock types using the same type of technique as described in the previous example. This has the advantage that if each coarse block is made up of one rock type only, one would not have to upscale the rock-fluid properties. On the other hand, this will typically lead to coarse grids with (highly) irregular block geometries and large variations in block volumes. To illustrate this point, we start by partitioning the grid uniformly into $6 \times 12 \times 3$ coarse blocks in index space:

```
p = partitionUI(G,[6 12 3]);
```

This will introduce a partition of all cells in the logical $40 \times 120 \times 20$ grid, including cells that are inactive. To get a contiguous partition vector, we remove blocks that contain no active cells, and then renumber the vector, which reduces the total number of blocks from 216 to 201. Some of the blocks may contain disconnected cells because of faults and other nonconformities, and we therefore need to postprocess the grid in physical space and split each disconnected block into a new set of connected sub-blocks:

```
p = compressPartition(p);
p = processPartition(G,p);
```

The result is a partition with 243 blocks, in which each coarse block consists of a set of connected cells in the fine grid. Figure 4.5 shows the individual coarse blocks using the explosion-view technique introduced above. Whereas all cells in the original model are almost exactly the same size, there is almost two orders difference between the volumes of the smallest and largest blocks in the coarsened model. In particular, the irregular boundary near the crest of the model will introduce small blocks that consist of only a single fine cell in the lateral direction. Large variations in block volumes will adversely affect any flow solver that is later run on the model and to get a more even size distribution for the coarse blocks, we will therefore remove these small blocks by merging them with the neighbor that has the smallest block volume. This is done repeatedly until the volumes of all blocks are above the lower threshold.

The merging algorithm is quite simple: we compute the block volumes, select the block with the smallest volume, and then merge this block with one of its neighbors. Then we update the partition vector by relabling all cells in the block with the new block number, compress the partition vector to get rid of empty entries, regenerate a coarse grid, recompute block volumes, pick the block with the smallest volume in the new grid, and so on. In each iteration, we plot the selected block and its neighbors:

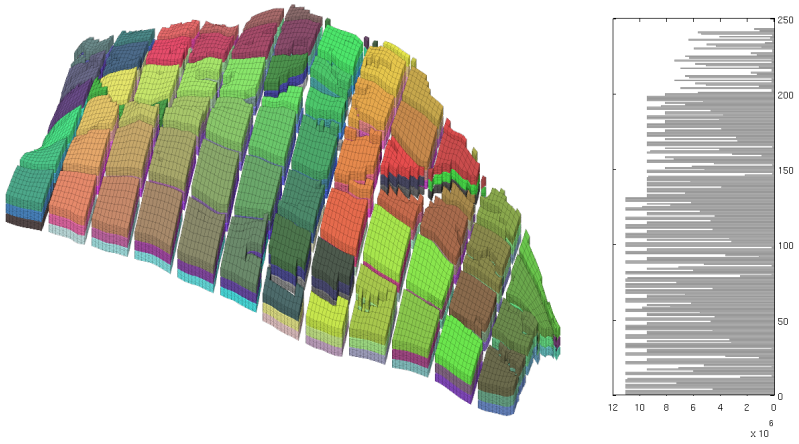


Fig. 4.5. Logically Cartesian partition of the SAIGUP model. The plot to the left shows the individual blocks in an explosion view using the `colorcube` colormap. The bar graph to the right shows the volumes in units [m^3] for each of the blocks in the partition.

```
blockVols = CG.cells.volumes;
meanVol  = mean(blockVols);
[minVol, block] = min(blockVols);
while minVol<.1*meanVol
  % Find all neighbors of the block
  clist = any(CG.faces.neighbors==block,2);
  nlist = reshape(CG.faces.neighbors(clist,:),[],1);
  nlist = unique(nlist(nlist>0 & nlist~=block));

  plotBlockAndNeighbors(CG, block, ...
     'PlotFaults', [false, true], 'Alpha', [1 .8 .8 .8]);

  % Merge with neighbor having largest volume
  [~,merge] = max(blockVols(nlist));

  % Update partition vector
  p(p==block) = nlist(merge);
  p = compressPartition(p);

  % Regenerate coarse grid and pick the block with the smallest volume
  CG = generateCoarseGrid(G, p);
  CG = coarsenGeometry(CG);
  blockVols = CG.cells.volumes;
  [minVol, block] = min(blockVols);
end
```

To find the neighbors of a given block, we first select all connections that involve block number `block`, which we store in the logical mask `clist`. We then extract the indices of the blocks involved in these connections by using `clist` to index the connection list `CG.faces.neighbors`. The block cannot be merged with the exterior or itself, so the values 0 and `block` are filtered out. In general, there may be more than one connection between a pair of blocks, so as a last step we use `unique` to remove multiple occurrences of the same block number.

When selecting which neighbor a block should be merged with, there are several points to consider from a numerical point of view. We will typically want to keep the blocks as regular and equally sized as possible, make sure that the cells inside each new block are well connected, and limit the number of new connections we introduce between blocks.

Figure 4.6 shows several of the small blocks and their neighbors. In the algorithm above, we have chosen a simple merging criterion: each block is merged with the neighbor having the largest volume. In iterations one and three, the blue blocks will be merged with the cyan blocks, which is probably fine in both cases. However, if the algorithm later wants to merge the yellow blocks, using the same approach may not give good results as shown in iteration ten. Here, it is natural to merge the blue block with the cyan block that lies in the same geological layer (same K index) rather than merging it with the magenta block that has the largest volume. The same problem is seen in iteration number four, where the blue block is merged with the yellow block, which is in another geological layer and only weakly connected to the blue block. As an alternative, we could choose to merge with the neighbor having the smallest volume, in which case the blue block would be merged with the yellow block in iterations one and three, with the magenta block in iteration four, and with the cyan block in iteration six. This would tend to create blocks consisting of cells from a single column in the fine grid, which may not be what we want.

Altogether, the figure and the discussion illustrate that there are mutually conflicting criteria that makes it difficult to merge blocks in the grid in a simple and robust manner. A more advanced strategy could, for instance, include additional constrains that penalize merging blocks belonging to different layers in the coarse grid. Likewise, one may want to avoid to create connections that only involve small surface areas. However, such connections may already be present in the coarsening we start from, as shown in the lower-right plot in Figure 4.6. Here, the yellow and cyan blocks were created when partitioning the grid uniformly in index space. The basic `processPartition` routine only checks that there is a connection between all cells inside a block. A more sophisticated routine could obviously also check the size of the face areas associated with each connection and consider different parts of the block to be disconnected if the area that connects them is too small. As a step in this direction, we could consider the face are when we postprocess the first uniform partition, e.g., do something like the following,

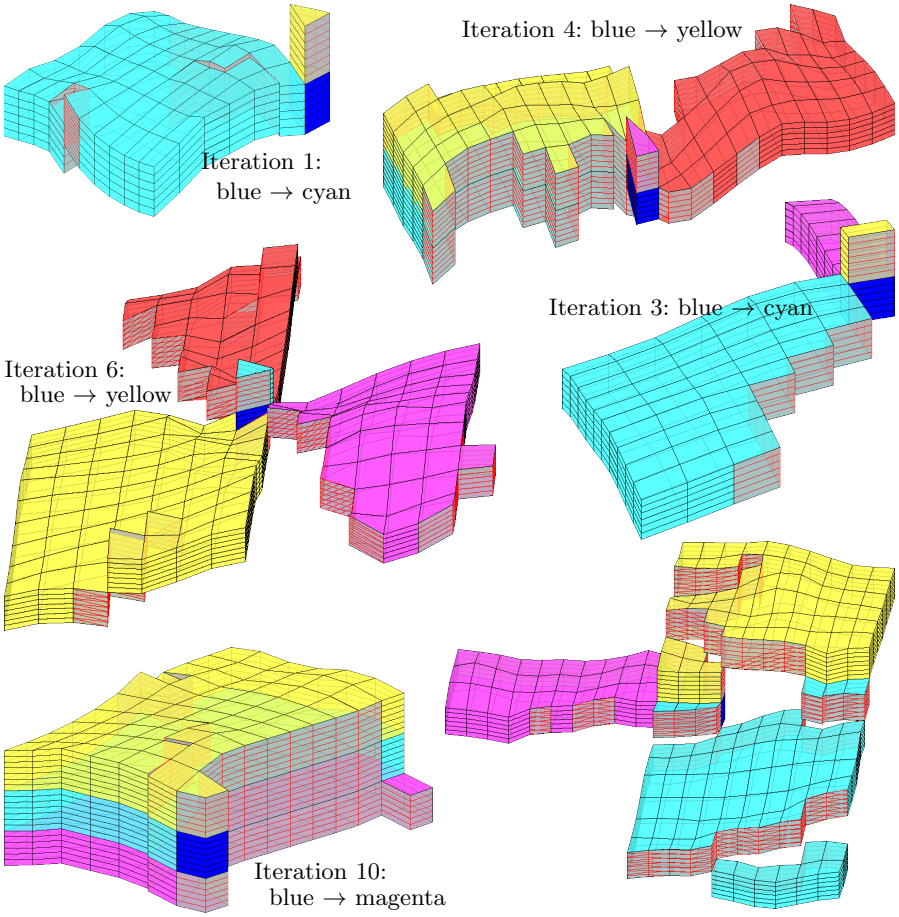


Fig. 4.6. Examples of small blocks (in blue) being merged with one of their neighbors shown in semi-transparent color with fault faces in gray.

```
p = partitionUI(G,[6 12 3]);
p = compressPartition(p);
p = processPartition(G, p, G.faces.areas<250);
```

This will increase the number of blocks in the final grid, after merging small blocks, from 220 to 273, but will avoid constructing blocks looking like the yellow and cyan block in the lower-right plot in Figure 4.6. On the other hand, this approach will obviously involve a threshold parameter that will vary from case to case and will need to be set by an expert. The result may also be very sensitive to the choice of this parameter. To see this, you can try to redo the initial partition with threshold values 300 and 301 for the face areas.

4.4 General Advice and Simple Guidelines

In most cases, the coarse partition will be used as input to some kind of flow simulation, which we will continue to discuss in the rest of the book. We conclude the discussion of grid coarsening by suggesting some simple and conceptual guidelines that we think can serve as a good starting point if you want to develop your own custom partitions. The guidelines refer to the left plot in Figure 4.7 for illustrations:

1. The partition should preferably minimize the occurrence of bidirectional flow across coarse-grid interfaces. Examples of grid structures that increase the likelihood for bidirectional flow are:
   - Coarse-grid faces with (highly) irregular shapes, like the 'saw-tooth' faces between Blocks 6 and 7 and Blocks 3 and 8.
   - Blocks that have only one neighbor, like Block 4 (unless the block contains source terms). A simple remedy for this is to split the interface into at least two sub-faces, and define a basis function for each sub-face.
   - Blocks having interfaces only along and not transverse to the major flow directions, like Block 5. To represent flow in a certain direction, there must be at least one non-tangential face that defines a basis function in the given flow direction.
2. Blocks and faces in the coarse grid should follow geological layers whenever possible. This is not fulfilled for Blocks 3 and 8.
3. Blocks in the coarse-grid should adapt to flow obstacles (shale barriers, etc.) whenever possible.

In addition, to enhance the efficiency of a simulator, one should try to keep the number of connections between coarse-grid blocks as low as possible (to minimize the bandwidth of the discretized systems), and avoid having too many small blocks which will increase the dimension of the discrete system

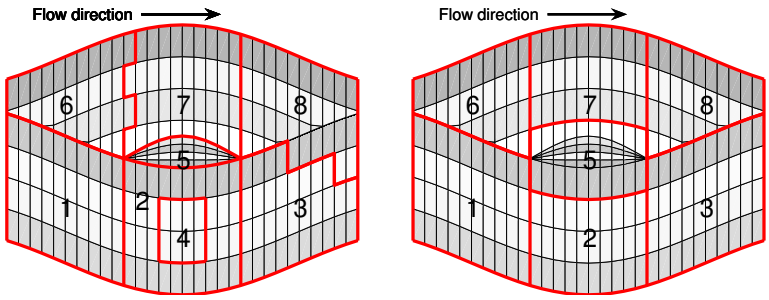


Fig. 4.7. Illustration of some of the guidelines for choosing a good coarse grid. In the left plot, all blocks except for Block 1 violate at least one of the guidelines each. In the right plot, the blocks have been improved at the expense of an increased number of connections.

and adversely affect its stability without necessarily improving the accuracy significantly.

In the right plot of Figure 4.7, we have used the guidelines above to improve the coarse grid from the left plot. In particular, we have increased the size of Block 5 to homogenise the block volumes and introduce basis functions in the major flow direction for this block. In doing so, we increase the number of couplings from nine to twelve (by removing the coupling between Blocks 2 and 4 and introducing extra coupling among Blocks 1, 3, 5, 6, and 8). In general, it may be difficult to obtain an 'optimal' coarse grid, since guidelines may be in conflict with each other. Indeed, having worked with coarsening algorithms for many years, it is our experience that the criteria that determine whether a coarsening is good or not will to a large degree depend on what purpose the grid is to be used for after coarsening. In particular, the coarsening strategy should reflect the type of process one wants to simulate and how the flow process is affected by the petrophysical properties the grid is populated with. To explain this, we will give a few examples:

- For a strongly heterogeneous system in which the fluid flow will mainly be dictated by the media properties (think of the Upper Ness formation from the SPE10 model in Section 2.4.3), a geometrically complex partition that adapts to contrasts in the media properties will most likely give better accuracy than a regular (Cartesian-like) partition in which individual blocks contain strong media contrasts. On the other hand, for media with small or smooth variations in the petrophysical parameters (think of a homogeneous medium, the Tarbert formation from SPE10, or something in between), one might be better off with a regular partition that follows the axial directions to minimize the introduction of geometrical artifacts in the numerical discretization.
- If your grid model contains different rock types (think of the SAIGUP model shown in Figure 2.16), you may want the partition to adapt to interfaces between these rock types if they correspond to significantly different petrophysical or rock-fluid properties. If they do not, creating an adapted grid will only introduce unnecessary geometrical complexities.
- If you study a fluid system in which one fluid is much lighter than the other fluid(s), as is the case when simulating gas injection or CO_2 sequestration, you may want to have high grid resolution near the top of your model to be able to capture the lighter fluid's tendency to override the other fluid(s) or retain the vertical resolution of the original grid to accurately resolve the gravity segregation in the fluid system.

In summary, it is our general and perhaps somewhat disappointing observation that making a good coarse grid is more an art than an exact science. As a result, MRST does not provide well-defined workflows for coarsening, but rather offers tools that (hopefully) are sufficiently flexible to support you in combining your creativity, physical intuition, and experience to generate good coarse grids.

Part II

Single-Phase Flow

5

Mathematical Models and Basic Discretizations

If you have read the chapters of the book in chronological order, you have already encountered the equations modeling flow of a single, incompressible fluid through a porous media twice: first in Section 1.3 where we showed how to use MRST to compute vertical equilibrium inside a gravity column, and then in Section 2.3.2, in which we discussed the concept of rock permeability. In this section, we will review the mathematical modeling of single-phase flow in more detail, introduce basic numerical methods for solving the resulting equations, and discuss how these are implemented in MRST and can be combined with the tools introduced in Chapters 2 and 3 to develop efficient simulators for single-phase incompressible flow. Solvers for compressible flow will be discussed in more detail in Chapter 7.

5.1 Fundamental concept: Darcy's law

Mathematical modeling of single-phase flow in porous media started with the work of Henry Darcy, a french hydraulic engineer, who in the middle of the 19th century was engaged by to enlarge and modernize the waterworks of the city of Dijon. To understand the physics of flow through the sand filters that were used to clean the water supply, Darcy designed a vertical experimental tank filled with sand, in which water was injected at the top and allowed to flow out at the bottom of the tank; Figure 5.1 shows a conceptual illustration. Once the sand pack is filled with water, and the inflow and outflow rates are equal, the hydraulic head at the inlet and at the outlet can be measured using mercury-filled manometers. The hydraulic head is given as, $h = E/mg = z+p/\rho g$, relative to a fixed datum. As water flows through the porous medium, it will experience a loss of energy. In a series of experiments, Darcy measured the water volumetric flow rate out of the tank and compared this rate with the loss of hydrostatic head from top to bottom of the column. From the experiments, he established that for the same sand pack, the discharge (flow rate) Q [m^3/s] is proportional to the cross-sectional area A [m^2], proportional

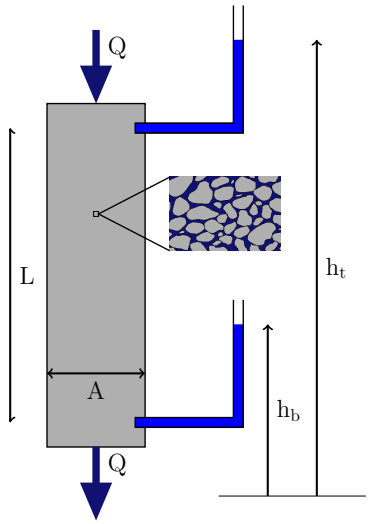


Fig. 5.1. Conceptual illustration of Darcy's experiment.

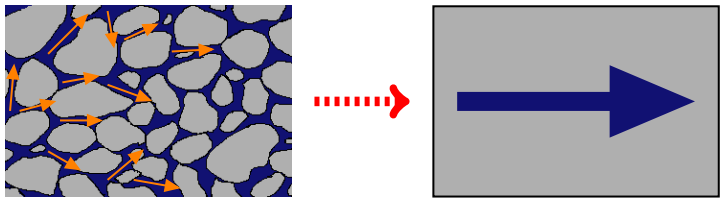


Fig. 5.2. The macroscopic Darcy velocity represents an average of microscopic fluid fluxes.

to the difference in hydraulic head (height of the water) $h_t - h_b$ [m], and inversely proportional to the flow length of the tank L [m]. Altogether, this can be summarized as

$$\frac{Q}{A} = \kappa \frac{h_t - h_b}{L} \tag{5.1}$$

which was presented in 1856 as an appendix to [21] entitled "Determination of the laws of flow of water through sand" and is what we today call Darcy's law. In (5.1), κ [m/s] denotes the hydraulic conductivity, which is a function both of the medium and the fluid flowing through it. It follows from a dimensional analysis that $\kappa = \rho g K/\mu$, where g [m/s^2] is the gravitational acceleration, μ [kg/ms] is the dynamic viscosity, and K [m^2] is the intrinsic permeability of a given sand pack.

The specific discharge $v = Q/A$, or Darcy flux, through the sand pack represents the volume of fluid per total area per time and has dimensions [m/s]. Somewhat misleading, v is often referred to as the Darcy velocity. However, since only a fraction of the cross-sectional area is available for flow (the major-

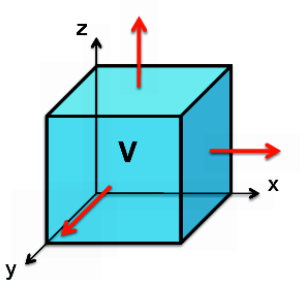


Fig. 5.3. Illustration of a control volume Ω on which one can apply the principle of conservation to derive macroscopic continuity equations.

ity of the area is blocked by sand grains), v is not a velocity in the microscopic sense. Instead, v is the apparent macroscopic velocity obtained by averaging the microscopic fluxes inside representative elementary volumes (REVs) which were discussed in Section 2.2.2. The macroscopic fluid velocity, defined as volume per area occupied by fluid per time, is therefore given by v/ϕ, where ϕ is the porosity associated with the REV.

Henceforth, we will, with a slight abuse of notation, refer to the specific discharge as the Darcy velocity. In modern differential notation, Darcy's law for a single-phase fluid reads,

$$\vec{v} = \frac{\mathsf{K}}{\mu}(\nabla p - g\rho\nabla z), \tag{5.2}$$

where p is the fluid pressure and z is the vertical coordinate. The equation expresses conservation of momentum and was derived from the Navier–Stokes equations by averaging and neglecting inertial and viscous effects by Hubbert in 1956. The observant reader will notice that Darcy's law 5.2 is analogous to Fourier's law (1822) for heat conduction, Ohm's law (1827) in the field of electrical networks, or Fick's law (1855) for fluid concentrations in diffusion theory, except that for Darcy there are two driving forces, pressure and gravity. Notice also that Darcy's law assumes a reversible fluid process, which is a special case of the more general physical laws of irreversible processes that were first described by Onsager.

5.2 General flow equations for single-phase flow

To derive a mathematical model for single-phase flow on the macroscopic scale, we first make a continuum assumption based on the existence of representative elementary volumes as discussed in the previous section and then look at a control volume as shown in Figure 5.3. From the fundamental law of mass conservation, we know that the accumulation of mass inside this volume must equal the net flux over the boundaries,

$$\frac{\partial}{\partial t}\int_\Omega \phi\rho\, d\vec{x} + \int_{\partial\Omega} \rho\vec{v}\cdot\vec{n}\, ds = \int_\Omega q\, d\vec{x}, \tag{5.3}$$

where ρ is the density of the fluid, ϕ is the rock porosity, $\vec{v}$ is the macroscopic Darcy velocity, $\vec{n}$ denotes the normal at the boundary $\partial\Omega$ of the computational domain Ω, and q denotes fluid sources and sinks, i.e., outflow and inflow of fluids per volume at certain locations. Applying Gauss' theorem, this conservation law can be written on the alternative integral form

$$\int_\Omega \left[\frac{\partial}{\partial t}\phi\rho + \nabla\cdot(\rho\vec{v})\right] d\vec{x} = \int_\Omega q\, d\vec{x}. \tag{5.4}$$

This equation is valid for any volume Ω, and in particular volumes that are infinitesimally small, and hence it follows that the macroscopic behavior of the single-phase fluid must satisfy the continuity equation

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla\cdot(\rho\vec{v}) = q. \tag{5.5}$$

Equation (5.5) contains more unknowns than equations and to derive a closed mathematical model, we need to introduce what is commonly referred to as constitutive equations that give the relationship between different states of the system (pressure, volume, temperature, etc) at given physical conditions. Darcy's law, discussed in the previous section, is an example of a constitutive relation that has been derived to provide a phenomenological relationship between the macroscale $\vec{v}$ and the fluid pressure p. In Section 2.3.1 we introduced the rock compressibility $c_r = d\ln(\phi)/dp$, which describes the relationship between the porosity ϕ and the pressure p. In a similar way, we can introduce the fluid compressibility to relate the density ρ to the fluid pressure p.

A change in density will generally cause a change in both the pressure p and the temperature T. The usual way of describing these changes in thermodynamics is to consider the change of volume V for a fixed number of particles,

$$\frac{dV}{V} = \frac{1}{V}\left(\frac{\partial V}{\partial p}\right)_T dp + \frac{1}{V}\left(\frac{\partial V}{\partial T}\right)_p dT, \tag{5.6}$$

where the subscripts T and p indicate that the change takes place under constant temperature and pressure, respectively. Since ρV is constant for a fixed number of particles, $d\rho V = \rho dV$, and (5.6) can written in the equivalent form

$$\frac{d\rho}{\rho} = \frac{1}{\rho}\left(\frac{\partial\rho}{\partial p}\right)_T dp + \frac{1}{\rho}\left(\frac{\partial\rho}{\partial T}\right)_p dT = c_f dp + \alpha_f dT, \tag{5.7}$$

where the c_f denotes the *isothermal compressibility* and α_f denotes the *thermal expansion coefficient*. In many subsurface systems, the density changes slowly so that heat conduction keeps the temperature constant, in which case (5.7) simplifies to

$$c_f = \frac{1}{\rho}\frac{d\rho}{dp} = \frac{d\ln(\rho)}{dp}. \tag{5.8}$$

The factor c_f, which we henceforth will refer to as the fluid compressibility, is non-negative and will generally depend on both pressure and temperature, i.e., $c_f = c_f(p, T)$.

Introducing Darcy's law and fluid and rock compressibilities in (5.5), we obtain the following parabolic equation for the fluid pressure

$$c_t \phi \rho \frac{\partial p}{\partial t} - \nabla \cdot \big[\frac{\rho \mathsf{K}}{\mu}(\nabla p - g\rho\nabla z)\big] = q, \tag{5.9}$$

where $c_t = c_r + c_f$ denotes the total compressibility. Notice that this equation is generally *nonlinear* since both ρ and c_t may depend on p. In the following, we will look briefly at several special cases in which the governing single-phase equation becomes a linear equation for the primary unknown; more extensive discussions can be found in standard textbooks like [50, Chap. 1], [18, Chap. 2]. For completeness, we will also briefly review the concept of an equation-of-state.

Incompressible flow

In the special case of an incompressible rock and fluid (that is, ρ and ϕ are independent of p so that $c_t = 0$), (5.9) simplifies to an elliptic equation with variable coefficients,

$$\nabla \cdot \big[\frac{\rho \mathsf{K}}{\mu}\nabla(p - g\rho z)\big] = q. \tag{5.10}$$

If we introduce the fluid potential, $\Phi = p - g\rho z$, (5.10) can be recognized as the (generalized) Poisson's equation $\nabla \cdot \mathsf{K}\nabla\Phi = q$ or as the Laplace equation $\nabla \cdot \mathsf{K}\nabla\Phi = 0$ if there are no volumetric fluid sources or sinks. In the next section, we will discuss in detail how to discretize the second-order spatial Laplace operator $\mathcal{L} = \nabla \cdot \mathsf{K}\nabla$, which is a key technological component that will enter almost any software for simulation of flow in porous rock formations.

Constant compressibility

If the fluid compressibility is constant and independent of pressure, (5.8) can be integrated from a known density ρ_0 at a pressure datum p_0 to give the following equation,

$$\rho(p) = \rho_0 e^{c_f(p-p_0)} \tag{5.11}$$

which applies well to most liquids that do not contain large quantities of dissolved gas. To develop the differential equation, we first assume that the porosity and the fluid viscosity do not depend on pressure. Going back to the definition of fluid compressibility (5.8), it also follows from this equation that $\nabla p = (c_f \rho)^{-1}\nabla\rho$, which we can use to eliminate ∇p from Darcy's law (5.2). Inserting the result into (5.5) gives us the following continuity equation

$$\frac{\partial \rho}{\partial t} - \frac{1}{\mu\phi c_f}\nabla \cdot \big(\mathsf{K}\nabla\rho - cg\rho^2\mathsf{K}\nabla z\big) = q, \tag{5.12}$$

which in the absence of gravity forces and source terms is a linear equation for the fluid density that is similar to the classical heat equation with variable coefficients,

$$\frac{\partial \rho}{\partial t} = \frac{1}{\mu \phi c_f} \nabla \cdot \big(\mathsf{K} \nabla \rho\big). \tag{5.13}$$

Slightly compressible flow

In the case that the fluid compressibility is small, it is sufficient to use a linear relationship

$$\rho = \rho_0 \big[1 + c_f (p - p_0)\big]. \tag{5.14}$$

We further assume that ϕ is a function of $\vec{x}$ only and that μ is constant. For simplicity, we also assume that g and q are both zero. Then, we can simplify (5.9) as follows:

$$(c_f \phi \rho) \frac{\partial p}{\partial t} = \frac{c_f \rho}{\mu} \nabla p \cdot \mathsf{K} \nabla p + \frac{\rho}{\mu} \nabla \cdot (\mathsf{K} \nabla p)$$

If c_f is sufficiently small, in the sense that $c_f \nabla p \cdot \mathsf{K} \nabla p \ll \nabla \cdot (\mathsf{K} \nabla p)$, we can neglect the first term on the right-hand side to derive a linear equation similar to (5.13) for the fluid pressure

$$\frac{\partial p}{\partial t} = \frac{1}{\mu \phi c_f} \nabla \cdot \big(\mathsf{K} \nabla p\big). \tag{5.15}$$

Ideal gas

If the fluid is a gas, compressibility can be derived from the gas law, which for an ideal gas can be written in two alternative forms,

$$pV = nRT, \qquad \rho = p(\gamma - 1)e. \tag{5.16}$$

In the first form, T is temperature, V is volume, R is the gas constant (8.314 J $\mathrm{K}^{-1}\mathrm{mol}^{-1}$), and $n = m/M$ is the amount of substance of the gas in moles, where m is the mass and M is the molecular weight. In the second form, γ is the adiabatic constant, i.e., ratio of specific heats at constant pressure and constant volume, and e is the specific internal energy (internal energy per unit mass). In either case, it follows from (5.8) that $c_f = 1/p$.

If the fluid is a gas, we can neglect gravity, and once again we assume that ϕ is a function of $\vec{x}$ only. Inserting (5.16) into (5.9) gives

$$\frac{\partial(\rho\phi)}{\partial t} = \phi(\gamma - 1)e \frac{\partial p}{\partial t} = \frac{1}{\mu} \nabla \cdot \big(\rho \mathsf{K} \nabla p\big) = \frac{(\gamma - 1)e}{\mu} \nabla \cdot \big(p \mathsf{K} \nabla p\big)$$

from which it follows that

$$\phi\mu \frac{\partial p}{\partial t} = \nabla \cdot \big(p \mathsf{K} \nabla p\big) \qquad \Leftrightarrow \qquad \frac{\phi\mu}{p} \frac{\partial p^2}{\partial t} = \nabla \cdot \big(\mathsf{K} \nabla p^2\big). \tag{5.17}$$

Equation of state

Equations (5.11), (5.14), and (5.16) are all examples of what is commonly referred to as equations of state, which provide constitutive relationships between mass, pressures, temperature, and volumes at thermodynamic equilibrium. Another popular form of these equations are the so-called cubic equations of state, which can be written as cubic functions of the molar volume $V_m = V/n = M/\rho$ involving constants that depend on the pressure p_c, the temperature T_c, and the molar volume V_c at the critical point, i.e., the point at which $(\frac{\partial p}{\partial V})_T = (\frac{\partial^2 p}{\partial V^2})_T \equiv 0$. A few particular examples include the Redlich–Kwong–Soave equation of state

$$\begin{aligned} p &= \frac{RT}{V_m - b} - \frac{a\alpha}{\sqrt{T}\, V_m(V_m + b)}, \\ a &= \frac{0.427 R^2 T_c^2}{p_c}, \quad b = \frac{0.08664 R T_c}{p_c} \\ \alpha &= \left[1 + \left(0.48508 + 1.55171\omega - 0.15613\omega^2\right)\left(1 - \sqrt{T/T_c}\right)\right]^2 \end{aligned} \tag{5.18}$$

and the Peng–Robinson equation of state,

$$\begin{aligned} p &= \frac{RT}{V_m - b} - \frac{a\alpha}{V_m^2 + 2bV_m - b^2)}, \\ a &= \frac{0.4527235 R^2 T_c^2}{p_c}, \quad b = \frac{0.077796 R T_c}{p_c} \\ \alpha &= \left[1 + \left(0.37464 + 1.54226\omega - 0.26992\omega^2\right)\left(1 - \sqrt{T/T_c}\right)\right]^2 \end{aligned} \tag{5.19}$$

In both equations, ω denotes the acentric factor of the species, which is a measure of the centricity (deviation from spherical form) of the molecules in the fluid. The Peng–Robinson model is much better at predicting the densities of liquids than the Redlich–Kwong–Soave model, which was developed to fit pressure data of hydrocarbon vapor phases. If we introduce

$$A = \frac{a\alpha p}{(RT)^2}, \qquad B = \frac{bp}{RT}, \qquad Z = \frac{pV}{RT},$$

the Redlich–Kwong–Soave equation (5.18) and the Peng–Robinson equation (5.19) can be written in alternative polynomial forms,

$$0 = Z^3 - Z^2 + Z(A - B - B^2) - AB, \tag{5.20}$$

$$0 = Z^3 - (1 - B)Z^2 + (A - 2B - 3B^2)Z - (AB - B^2 - B^3), \tag{5.21}$$

which explain the why they are called cubic equations of state.

5.3 Auxiliary conditions and equations

The governing equations for single-phase flow discussed above are all parabolic equations, except for the incompressible case in which the governing equation

is elliptic. For the solution to be well-posed[1] inside a finite domain for any of the equations, one needs to supply boundary conditions that determine the behavior on the external boundary. For the parabolic equations describing unsteady flow, one also needs to impose an initial condition that determines the initial state of the fluid system. In this section, we will discuss these conditions in more detail. We will also discuss models for representing flow in and out of the reservoir rock through wellbores. Because this flow typically takes place on a length scale that is much smaller than the length scales of the global flow inside the reservoir, it is customary to model it using special analytical models. Finally, we also discuss a set of auxiliary equations for describing the movement of fluid elements and/or neutral particles that follow the single-phase flow without affecting it.

5.3.1 Boundary and initial conditions

In reservoir simulation one is often interested in describing closed flow systems that have no fluid flow across its external boundaries. This is a natural assumption when studying full reservoirs that have trapped and contained petroleum fluids for million of years. Mathematically, no-flow conditions across external boundaries are modeled by specifying homogeneous Neumann conditions,

$$\vec{v} \cdot \vec{n} = 0 \quad \text{for } \vec{x} \in \partial\Omega. \tag{5.22}$$

With no-flow boundary conditions, any pressure solution of (5.10) is immaterial and only defined up to an additive constant, unless a datum value is prescribed at some internal point or along the boundary.

It is also common that parts of the reservoir may be in communication with a larger aquifer system that provide external pressure support, which can be modeled in terms of a Dirichlet condition of the form

$$p(\vec{x}) = p_a(\vec{x}, t) \quad \text{for } \vec{x} \in \Gamma_a \subset \partial\Omega. \tag{5.23}$$

The function p_a can, for instance, be given as a hydrostatic condition. Alternatively, parts of the boundary may have a certain prescribed influx, which can be modeled in terms of an inhomogeneous Neumann condition,

$$\vec{v} \cdot \vec{n} = u_a(\vec{x}, t) \quad \text{for } \vec{x} \in \Gamma_a \subset \partial\Omega. \tag{5.24}$$

Combinations of these conditions are used when studying parts of a reservoir (e.g., a sector models). There are also cases, e.g., when describing groundwater systems or CO_2 sequestration in saline aquifers, where (parts of) the boundaries are open or the system contains a background flow. More information of how to set boundary conditions will be given in Section 6.1.4. In

[1] A solution is well-posed if it exists, is unique, and depends continuously on the initial and boundary conditions.

the compressible case in (5.9), we also need to specify an initial pressure distribution. Typically, this pressure distribution will be hydrostatic, as in the gravity column we discussed briefly in Section 1.3, and hence be given by the ordinary differential equation,

$$\frac{dp}{dz} = \rho g, \qquad p(z_0) = p_0. \tag{5.25}$$

5.3.2 Models for injection and production wells

In a typical reservoir simulator, the inflow and outflow in wells occur on a subgrid scale. Therefore, special models have to be developed to model this particular flow. Normally, fluids are injected in a grid block at either constant *surface rate* or at constant *bottom-hole pressure*, which is sometimes also called wellbore flowing pressure. Similarly, fluids are produced at constant bottom-hole pressure or constant surface liquid rate. The expression inflow performance relation (IPR) denotes the relation between the bottom-hole pressure and the surface flow rate.

The simplest and most widely used inflow-performance relation is the linear law

$$q_o = J(p_R - p_{bh}), \tag{5.26}$$

which states that the flow rate is directly proportional to the pressure drawdown in the well; that is, flow rate is proportional to the difference between the average reservoir pressure p_R in the grid cell and the bottom-hole pressure p_{bh} in the well. The constant of proportionality J is called the *productivity index* (PI) for production wells or the *well injectivity index* (WI) for injectors and accounts for all rock and fluid properties, as well as geometric factors that affect the flow. In MRST, we do not distinguish between productivity and injectivity indices, and henceforth we will only use the shorthand 'WI'.

The basic linear relation (5.26) can be derived from Darcy's law. Consider a vertical well that drains a rock with uniform permeability K. As an equation of state, we introduce the formation volume factor B defined as the ratio between the volume of the fluid at reservoir conditions and the volume of the fluid at surface conditions. (For incompressible flow, $B \equiv 1$). The well penetrates the rock completely a height h and is open in the radial direction. We assume a pseudo-steady, radial flow that can be described by Darcy's law

$$v = \frac{qB}{2\pi r h} = \frac{K}{\mu}\frac{dp}{dr}.$$

We now integrate this equation from the wellbore and to the drainage boundary $r = r_e$ where the pressure is constant

$$2\pi K h \int_{p_{bh}}^{p_e} \frac{1}{q\mu B}\, dp = \int_{r_w}^{r_e} \frac{1}{r}\, dr.$$

Here, B and μ are pressure-dependent quantities; B decreases with pressure and μ increases. The composite effect is that $(\mu B)^{-1}$ decreases (almost) linearly with pressure. We can therefore approximate μB by $(\mu B)_{avg}$ evaluated at the average pressure $p_{avg} = (p_{bh} + p_e)/2$. For convenience, we drop the subscript in the following. This gives us the pressure as a function of radial distance

$$p_e = p_{bh} + \frac{q\mu B}{2\pi Kh} \ln(r_e/r_w). \tag{5.27}$$

To close the system, we need to know the location of the drainage boundary $r = r_e$ where the pressure is constant. This is often hard to know, and it is customary to relate q to the volumetric average pressure instead. For pseudo-steady flow the volumetric average pressure occurs at $r = 0.472 r_e$. Hence,

$$q = \frac{2\pi Kh}{\mu B\big(\ln(r_e/r_w) - 0.75\big)}\big(p_R - p_{bh}\big). \tag{5.28}$$

The above relation (5.28) was developed for an ideal well under several simplified assumptions: homogeneous and isotropic formation of constant thickness, clean wellbore, etc. In practice, a well will rarely produce under these ideal conditions. Typically the permeability is altered close to the wellbore under drilling and completion, the well will only be partially completed, and so on. The actual pressure performance will therefore deviate from (5.28). To model this, it is customary to include a *skin factor* S to account for extra pressure loss due to alterations in the inflow zone. The resulting equation is

$$q = \frac{2\pi Kh}{\mu B\big(\ln(r_e/r_w) - 0.75 + S\big)}\big(p_R - p_{bh}\big). \tag{5.29}$$

Often the constant -0.75 is included in the skin factor S, and for stimulated wells the skin factor could be negative. Sometimes h is modified to ch, where c is the completion factor, i.e., a dimensionless number between zero and one describing the fraction of the wellbore open to flow.

To use the radial model in conjunction with a reservoir model, the volumetric average pressure in the radial model must be related to the computed cell average pressure. This was first done by Peaceman [51]. Assuming isotropic permeabilities, square grid blocks, single phase flow and a well at a center of an interior block, Peaceman showed that the equivalent radius is

$$r_e \approx 0.2\sqrt{\Delta x\, \Delta y}.$$

This basic model has later been extended to cover a lot of other cases, e.g., off-center wells, multiple wells, non-square grids, anisotropic permeability, horizontal wells; see for instance [9, 29, 5]. For anisotropic permeabilities–and horizontal wells–the equivalent radius is defined as [49]

$$r_e = 0.28 \frac{\Big(\sqrt{K_y/K_x}\Delta x^2 + \sqrt{K_x/K_y}\Delta y^2\Big)^{1/2}}{\big(K_y/K_x\big)^{1/4} + \big(K_x/K_y\big)^{1/4}}, \tag{5.30}$$

and the permeability is replaced by an effective permeability

$$K_e = \sqrt{K_x\, K_y}. \tag{5.31}$$

If we include gravity forces in the well and assume hydrostatic equilibrium, the well model thus reads

$$q_i = \frac{2\pi h c K_e}{\ln(r_e/r_w) + S} \frac{1}{\mu_i B_i} \big(p_R - p_{bh} - \rho_i (z - z_{bh}) g\big), \tag{5.32}$$

where K_e is given by (5.31) and r_e is given by (5.30). For deviated wells, h denotes the length of the grid block in the major direction of the wellbore and *not* the length of the wellbore.

5.3.3 Field lines and time-of-flight

Equation (5.10) together with a set of suitable and compatible boundary conditions is all that one needs to describe the flow of an incompressible fluid inside an incompressible rock. In the remains of this section, we will discuss a few simple concepts and auxiliary equations that have proven useful to visualize, analyze, and understand flow fields.

A simple way to visualize a flow field is to use field lines resulting from the vector field: streamlines, streaklines, and pathlines. In steady flow, the three are identical. However, if the flow is not steady, i.e., when $\vec{v}$ changes with time, they differ. Streamlines are associated with an instant snapshot of the flow field and consists of a family of curves that are everywhere tangential to $\vec{v}$ and show the direction a fluid element will travel at this specific point in time. That is, if $\vec{x}(r)$ is a parametric representation of a single streamline at this instance $\hat{t}$ in time, then

$$\frac{d\vec{x}}{dr} \times \vec{v}(\vec{x}, \hat{t}) = 0, \quad \text{or equivalently,} \quad \frac{d\vec{x}}{dr} = \frac{\vec{v}(\hat{t})}{|\vec{v}(\hat{t})|}. \tag{5.33}$$

In other words, streamlines are calculated instantaneously throughout the fluid from an *instantaneous* snapshot of the flow field. Because two streamlines from the same instance in time cannot cross, there cannot be flow across it, and if we align a coordinate along a bundle of streamlines, the flow through them will be one-dimensional.

Pathlines are the trajectories that individual fluid elements will follow over a certain period. In each moment of time, the path a fluid particle takes will be determined by the streamlines associated with the streamlines at this instance in time. If $\vec{y}(t)$ represents a single path line starting at $\vec{y}_0$ at time t_0, then

$$\frac{d\vec{y}}{dt} = \vec{v}(\vec{y}, t), \qquad \vec{y}(t_0) = \vec{y}_0. \tag{5.34}$$

A streakline is the line traced out by all fluid particles that have passed through a prescribed point throughout a certain period of time. (Think of dye

injected into the fluid at a specific point). If we $\vec{z}(t,s)$ denote a parametrization of a streakline and $\vec{z}_0$ the specific point through which all fluid particles have passed, then

$$\frac{d\vec{z}}{dt} = \vec{v}(\vec{z},t), \qquad \vec{z}(s) = \vec{z}_0. \tag{5.35}$$

Like streamlines, two streaklines cannot intersect each other.

In summary: streamline patterns change over time, but are easy to generate mathematically. Pathlines and streaklines are recordings of the passage of time and are obtained through experiments.

Within reservoir simulation streamlines are far more used that pathlines and streaklines. Moreover, rather than using the arc length r to parametrize streamlines, it is common to introduce an alternative parametrization called time-of-flight, which takes into account the reduced volume available for flow, i.e., the porosity ϕ. Time-of-flight is defined by the following integral

$$\tau(r) = \int_0^r \frac{\phi(\vec{x}(s))}{|\vec{v}(\vec{x}(s))|}\, ds, \tag{5.36}$$

where τ expresses the time it takes a fluid particle to travel a distance r along a streamline (in the interstitial velocity field $\vec{v}/\phi$). Alternatively, by the fundamental theorem of calculus and the directional derivative,

$$\frac{d\tau}{dr} = \frac{\phi}{|\vec{v}|} = \frac{\vec{v}}{|\vec{v}|} \cdot \nabla\tau,$$

from which it follows that τ can be expressed by the following differential equation [22, 23]

$$\vec{v} \cdot \nabla\tau = \phi. \tag{5.37}$$

In lack of a better name, we will refer to this as the time-of-flight equation.

5.3.4 Tracers and volume partitions

Somewhat simplified, tracers can be considered as neutral particles that passively flow with the fluid without altering its flow properties. The concentration of a tracer is given by a continuity equation on the same form as (5.5),

$$\frac{\partial(\phi C)}{\partial t} + \nabla \cdot \big(\vec{v}C\big) = q_C. \tag{5.38}$$

Communication patterns within a reservoir can be determined by simulating the evolution of artificial, neutral tracers. A simple flow diagnostics is to set the tracer concentration equal to one in a particular fluid source or at a certain part of the inflow boundary, and compute the concentration approached at steady-state conditions,

$$\nabla \cdot \big(\vec{v}C\big) = q_C, \qquad C|_{\text{inflow}} = 1. \tag{5.39}$$

The resulting tracer distribution gives the portion of the total fluid volume coming from a certain fluid source, or parts of the inflow boundary, that eventually will reach each point in the reservoir. Likewise, by reversing the sign of the flow field and assigning unit tracers to a particular fluid sink or parts of the outflow, one can compute the portion of the fluid arriving at a source or outflow boundary that can be attributed to a certain point in the reservoir. By repeating this process for all parts of the inflow, one can easily obtain a partition of the instantaneous flow field.

A more dynamic view can be obtained by utilizing the fact that streamlines and time-of-flight can be used to define an alternative curvilinear and flow-based coordinate system in three dimensions. To this end, we introduce the bi-streamfunctions ψ and χ [10], for which $\vec{v} = \nabla\psi \times \nabla\chi$. In the streamline coordinates (τ, ψ, χ), the gradient operator is expressed as

$$\nabla_{(\tau,\psi,\chi)} = (\nabla\tau)\frac{\partial}{\partial\tau} + (\nabla\psi)\frac{\partial}{\partial\psi} + (\nabla\chi)\frac{\partial}{\partial\chi}. \tag{5.40}$$

Moreover, a streamline Ψ is defined by the intersection of a constant value for ψ and a constant value for χ. Because $\vec{v}$ is orthogonal to $\nabla\psi$ and $\nabla\chi$, it follows from (5.37) that

$$\vec{v} \cdot \nabla_{(\tau,\psi,\chi)} = (\vec{v} \cdot \nabla\tau)\frac{\partial}{\partial\tau} = \phi\frac{\partial}{\partial\tau}. \tag{5.41}$$

Therefore, the coordinate transformation $(x, y, z) \to (\tau, \psi, \chi)$ will reduce the three-dimensional transport equation (5.38) to a family of one-dimensional transport equations along each streamline [22, 35], which for incompressible flow reads

$$\frac{\partial C}{\partial t} + \frac{\partial C}{\partial\tau} = 0. \tag{5.42}$$

In other words, there is no exchange of the quantity C between streamlines and each streamline can be viewed as an isolated flow system. Assuming a prescribed concentration history $C_0(t)$ at the inflow, gives a time-dependent boundary-value problem for the concentration at the outflow (5.42). Here, the response is given as [22],

$$C(t) = C_0(t - \tau), \tag{5.43}$$

which is easily verified by inserting the expression into (5.42) and the fact that the solution is unique [33]. For the special case of continuous and constant injection, the solution is particularly simple

$$C(t) = \begin{cases} 0, & t < \tau, \\ C_0, & t > \tau. \end{cases}$$

5.4 Basic finite-volume discretizations

Research on numerical solution of the Laplace/Poisson equation has a long tradition, and there exist a large number of different finite-difference and

finite-volume methods, as well as finite-element methods based on standard Galerkin, mixed, or discontinuous Galerkin formulations, which all have their merits. In Chapter 8, we will discuss consistent discretizations of Poisson-type equations in more detail. We introduce a general framework for formulating such method on general polyhedral grids and present several recent methods that are specially suited for irregular grids with strongly discontinuous coefficients, which are typically seen in realistic reservoir simulation models. In particular, we will discuss multipoint flux-approximation (MPFA) methods and mimetic finite-difference (MFD) methods, which are both available in add-on modules that are part of the standard MRST releases. As a starting point, however, we will in rest of this section present the simplest example of a finite-volume discretization, the two-point flux-approximation (TPFA) scheme, which is used extensively throughout industry and also is the default discretization method in MRST. We will give a detailed derivation of the method and point out its advantages and shortcomings. For completeness, we also briefly outline how to discretize the time-of-flight and the stationary tracer equations.

5.4.1 A two-point flux-approximation (TPFA) method

To keep technical details at a minimum, we will in the following without loss of generality consider the simplified single-phase flow equation

$$\nabla \cdot \vec{v} = q, \qquad \vec{v} = -\mathsf{K}\nabla p, \qquad \text{in } \Omega \subset \mathbb{R}^d. \tag{5.44}$$

In classical finite-difference methods, partial differential equations are approximated by replacing the derivatives with appropriate divided differences between point-values on a discrete set of points in the domain. Finite-volume methods, on the other hand, have a more physical motivation and are derived from conservation of (physical) quantities over cell volumes. Thus, in a finite-volume method the unknown functions are represented in terms of average values over a set of finite-volumes, over which the integrated PDE model is required to hold in an averaged sense. Although finite-difference and finite-volume methods have fundamentally different interpretation and derivation, the names are used interchangeably in the scientific literature. The main reason for this is probably that for certain low-order methods, the discrete equations derived for the cell-centered values in a mass-conservative finite-difference method are identical to the discrete equations for the cell averages in the corresponding finite-volume method. Herein, we will stick to this convention and not make a strict distinction between the two types of methods

To develop a finite-volume discretization for (5.44), we start by rewriting the equation in integral form (see (5.3)) using a single cell Ω_i in the discrete grid as control volume

$$\int_{\partial\Omega_i} \vec{v} \cdot \vec{n}\, ds = \int_{\Omega_i} q\, d\vec{x} \tag{5.45}$$

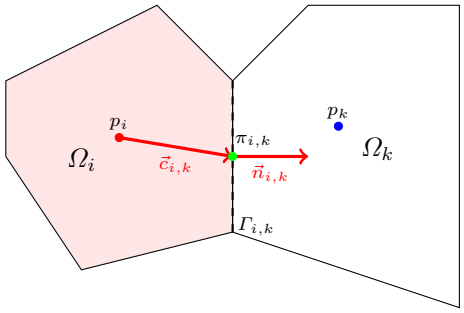


Fig. 5.4. Two cells used to define the two-point finite-volume discretization of the Laplace operator.

This equation will ensure that mass is conserved for each grid cell. The next step is to use Darcy's law to compute the flux across each face of the cell,

$$v_{i,k} = \int_{\Gamma_{ik}} \vec{v} \cdot \vec{n}\, ds, \qquad \Gamma_{i,k} = \partial\Omega_i \cap \partial\Omega_k. \tag{5.46}$$

We will refer to the faces $\Gamma_{i,k}$ as *half-faces* since they are associated with a particular grid cell Ω_i and a certain normal vector $\vec{n}_{i,k}$. However, since the grid is assumed to be matching, each interior half face will have a twin half-face $\Gamma_{k,i}$ that has identical area $A_{k,i} = A_{i,k}$ but opposite normal vector $\vec{n}_{k,i} = -\vec{n}_{i,k}$. If we further assume that the integral over the cell face in (5.46) is approximated by the midpoint rule, we use Darcy's law to write the flux as

$$v_{i,k} \approx A_{i,k}\vec{v}(\vec{x}_{i,k}) \cdot \vec{n}_{i,k} = -A_{i,k}\big(\mathsf{K}\nabla p\big)(\vec{x}_{i,k}) \cdot \vec{n}_{i,k}, \tag{5.47}$$

where $\vec{x}_{i,k}$ denotes the centroid on $\Gamma_{i,k}$. The idea is now to use a one-sided finite difference to express the pressure gradient as the difference between the pressure $\pi_{i,k}$ at the face centroid and at some point inside the cell. However, in a finite-volume method, we only know the cell averaged value of the pressure inside the cell. We therefore must make some additional assumption that will enable us to reconstruct point values that are needed to estimate the pressure gradient in Darcy's law. If we assume that the pressure is linear (or constant) inside each cell, the reconstructed value pressure value π_i at the cell center is identical to the average pressure p_i inside the cell, and hence it follows that (see Figure 5.4)

$$v_{i,k} = A_{i,k}\mathsf{K}_i \frac{(p_i - \pi_{i,k})\vec{c}_{i,k}}{|\vec{c}_{i,k}|^2} \cdot \vec{n}_{i,k} = T_{i,k}(p_i - \pi_{i,k}). \tag{5.48}$$

Here, we have introduced one-sided transmissibilities $T_{i,k}$ that are associated with a single cell and gives a two-point relation between the flux across a cell face and the difference between the pressure at the cell and face centroids. We will refer to these one-sided transmissibilities as *half-transmissibilities* since they are associated with a half face.

To derive the final discretization, we impose continuity of fluxes across all faces, $v_{i,k} = -v_{k,i} = v_{ik}$ and continuity of face pressures $\pi_{i,k} = \pi_{k,i} = \pi_{ik}$. This gives us two equations,

$$T_{i,k}^{-1} v_{ik} = p_i - \pi_{ik}, \qquad -T_{k,i}^{-1} v_{ik} = p_k - \pi_{ik}.$$

By eliminating the interface pressure π_{ik}, we end up with the following two-point flux-approximation (TPFA) scheme,

$$v_{ik} = \left[T_{i,k}^{-1} + T_{k,i}^{-1}\right]^{-1} (p_i - p_k) = T_{ik}(p_i - p_k). \tag{5.49}$$

where is the T_{ik} the transmissibility associated with the connection between the two cells. As the name suggests, the TPFA scheme uses two 'points', the cell averages p_i and p_k, to approximate the flux across the interface Γ_{ik} between the cells Ω_i and Ω_k. In the derivation above, the cell fluxes were parametrized in terms of the index of the neighboring cell. Extending the derivation to also include fluxes on exterior faces is trivial since we either know the flux explicitly for Neumann boundary conditions (5.22) or (5.24), or know the interface pressure for Dirichlet boundary conditions (5.23).

By inserting the expression for v_{ik} into (5.45), we see that the TPFA scheme for (5.44), in compact form, seeks a set of cell averages that satisfy the following system of equations

$$\sum_k T_{ik}(p_i - p_k) = q_i, \qquad \forall \Omega_i \subset \Omega \tag{5.50}$$

This system is clearly symmetric, and a solution is, as for the continuous problem, defined up to an arbitrary constant. The system is made positive definite, and symmetry is preserved by specifying the pressure in a single point. In MRST, we have chosen to set $p_1 = 0$ by adding a positive constant to the first diagonal of the matrix $\mathbf{A} = [a_{ij}]$, where:

$$a_{ij} = \begin{cases} \sum_k T_{ik} & \text{if } j = i, \\ -T_{ij} & \text{if } j \neq i, \end{cases}$$

The matrix $\mathbf{A}$ is sparse and will have a banded structure for structured grids (tridiagonal for 1D grids and penta- and heptadiagonal for logically Cartesian grids in 2D and 3D, respectively). The TPFA scheme is monotone, robust, and relatively simple to implement, and is currently the industry standard with reservoir simulation.

Example 5.1. To tie the links with standard finite-difference methods on Cartesian grids, we will derive the two-point discretization for a 2D Cartesian grid with isotropic permeability. Consider the flux in the x-direction between two cells i and k as illustrated in Figure 5.5. As above, we impose mass conservation inside each cell. For cell i this reads:

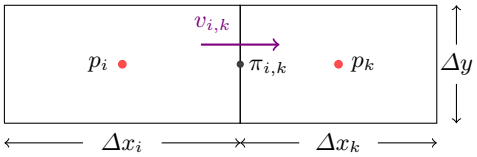


Fig. 5.5. Two cells used to derive the TPFA discretization for a 2D Cartesian grid

$$v_{i,k} = \Delta y \frac{(p_i - \pi_{i,k})}{\left(\frac{1}{2}\Delta x_i\right)^2}\left(\tfrac{1}{2}\Delta x_i, 0\right) K_i (1,0)^T = \Delta y \frac{2K_i}{\Delta x_i}(p_i - \pi_{i,k})$$

and likewise for cell k:

$$v_{k,i} = \Delta y \frac{(p_k - \pi_{k,i})}{\left(\frac{1}{2}\Delta x_k\right)^2}\left(-\tfrac{1}{2}\Delta x_k, 0\right) K_k (-1,0)^T = \Delta y \frac{2K_k}{\Delta x_k}(p_k - \pi_{k,i})$$

Next, we impose continuity of fluxes and face pressures,

$$v_{i,k} = -v_{k,i} = v_{ik}, \qquad \pi_{i,k} = \pi_{k,i} = \pi_{ik}$$

which gives us two equations

$$\frac{\Delta x_i}{2K_i \Delta y} v_{ik} = p_i - \pi_{ik}, \qquad -\frac{\Delta x_k}{2K_k \Delta y} v_{ik} = p_k - \pi_{ik}.$$

Finally, we eliminate π_{ik} to obtain

$$v_{ik} = 2\Delta y \left(\frac{\Delta x_i}{K_i} + \frac{\Delta x_k}{K_k}\right)^{-1}(p_i - p_k),$$

which shows that the transmissibility is given by the harmonic average of the permeability values in the two adjacent cells, as one would expect.

In [2], we showed how one could develop an efficient and self-contained MATLAB program that in approximately thirty compact lines solved the incompressible flow equation (5.44) using the two-point method outlined above. The program was designed for Cartesian grids with no-flow boundary conditions only and relied strongly on a logical ijk numbering of grid cells. For this reason, the program has limited applicability beyond highly idealized cases like the SPE10 model. However, in its simplicity, it presents an interesting contrast to the general-purpose implementation in MRST that handles unstructured grids, wells, and more general boundary conditions. The interested reader is encouraged to read the paper and try the accompanying program and example scripts that can be downloaded from

http://folk.uio.no/kalie/matlab-ressim/

5.4.2 Abstract formulation: discrete `div` and `grad` operators

While the double-index notation $v_{i,k}$ and v_{ik} used in the previous section is simple and easy to comprehend when working with a single interface between two neighboring cells, it becomes more involved when we want to introduce the same type of discretizations for more complex equations than the Poisson equation for incompressible flow. To prepare for discussions that will follow later in the book, we will in the following introduce a more abstract way of writing the two-point finite-volume discretization introduced in the previous section. The idea is to introduce discrete operators for the divergence and gradient operators that mimic their continuous counterparts, which will enable us to write the discretized version of the Poisson equation (5.44) in the same form as its continuous counterpart. To this end, we start by a quick recap of the definition of unstructured grids. As discussed in detail in Section 3.4, the grid structure in MRST, consists of three objects: The *cells*, the *faces*, and the *nodes*. Each cell corresponds to a set of faces, and each face to a set of *edges*, which again are determined by the nodes. Each object has given geometrical properties (volume, areas, centroids). As before, let us denote by n_c and n_f, the number of cells and faces, respectively. To define the topology of the grid, we will mainly use two different mappings. The first mapping is given by $N : \{1, \ldots, n_c\} \to \{0, 1\}^{n_f}$ and maps a cell to the set of faces that constitute this cell. In a grid structure `G`, this is represented as the `G.cells.faces` array, where the first column that gives the cell numbers is not stored since it is redundant and instead must be computed by a call `f2cn = gridCellNo(G);`. The second mapping consists in fact of two mappings that, for a given face, give the corresponding neighboring cells, $N_1, N_2 : \{1, \ldots, n_f\} \to \{1, \ldots, n_c\}$. In a grid structure `G`, N_1 is given by `G.faces.neighbors(:,1)` and N_2 by `G.faces.neighbors(:,2)`.

Let us now construct the discrete versions of the divergence and gradient operators, which we denote `div` and `grad`. The mapping `div` is a linear mapping from faces to cells. We consider a discrete flux $\boldsymbol{v} \in \mathbb{R}^{n_f}$. For a face f, the orientation of the flux $\boldsymbol{v}[f]$ is from $N_1(f)$ to $N_2(f)$. Hence, the total amount of matter leaving the cell c is given by

$$\texttt{div}(\boldsymbol{v})[c] = \sum_{f \in N(c)} \boldsymbol{v}[f]\, \mathbf{1}_{\{c=N_1(f)\}} - \sum_{f \in N(c)} \boldsymbol{v}[f]\, \mathbf{1}_{\{c=N_2(f)\}}. \tag{5.51}$$

The `grad` mapping maps $\mathbb{R}^{n_c}$ to $\mathbb{R}^{n_f}$ and it is defined as

$$\texttt{grad}(\boldsymbol{p})[f] = \boldsymbol{p}[N_2(f)] - \boldsymbol{p}[N_1(f)], \tag{5.52}$$

for any $\boldsymbol{p} \in \mathbb{R}^{n_c}$. In the continuous case, the gradient operator is the adjoint of the divergence operator (up to a sign), as we have

$$\int_\Omega p\, \nabla \cdot \vec{v}\, d\vec{x} + \int_\Omega \vec{v} \cdot \nabla p\, d\vec{x} = 0, \tag{5.53}$$

for vanishing boundary conditions. Let us prove that this property holds also in the discrete case. To simplify the notations, we set $S_c = \{1, \dots, n_c\}$ and $S_f = \{1, \dots, n_f\}$. For any $\boldsymbol{v} \in \mathbb{R}^{n_f}$ and $\boldsymbol{p} \in \mathbb{R}^{n_c}$, we have

$$\begin{aligned}\sum_{c\in S_c} \mathtt{div}(\boldsymbol{v})[c]\,\boldsymbol{p}[c] &= \sum_{c\in S_c} \boldsymbol{p}[c]\left(\sum_{f\in N(c)} \boldsymbol{v}[f]\,\mathbf{1}_{\{c=N_1(f)\}} - \sum_{f\in N(c)} \boldsymbol{v}[f]\,\mathbf{1}_{\{c=N_2(f)\}}\right)\\ &= \sum_{c\in S_c}\sum_{f\in S_f} \boldsymbol{v}[f]\,\boldsymbol{p}[c]\,\mathbf{1}_{\{c=N_1(f)\}}\,\mathbf{1}_{\{f\in N(c)\}}\\ &\quad - \sum_{c\in S_c}\sum_{f\in S_f} \boldsymbol{v}[f]\,\boldsymbol{p}[c]\,\mathbf{1}_{\{c=N_2(f)\}}\,\mathbf{1}_{\{f\in N(c)\}} \qquad (5.54)\end{aligned}$$

We can switch the order in the sums above and obtain

$$\sum_{c\in S_c}\sum_{f\in S_f} \boldsymbol{v}[f]\,\boldsymbol{p}[c]\,\mathbf{1}_{\{c=N_1(f)\}}\,\mathbf{1}_{\{f\in N(c)\}} = \sum_{f\in S_f}\sum_{c\in S_c} \boldsymbol{v}[f]\,\boldsymbol{p}[c]\,\mathbf{1}_{\{c=N_1(f)\}}\,\mathbf{1}_{\{f\in N(c)\}}.$$

For a given face f, we have that $\mathbf{1}_{\{c=N_1(f)\}}\mathbf{1}_{\{f\in N(c)\}}$ is nonzero if and only if $c = N_1(f)$ and therefore

$$\sum_{f\in S_f}\sum_{c\in S_c} \mathbf{1}_{\{c=N_1(f)\}}\mathbf{1}_{\{f\in N(c)\}}\boldsymbol{v}[f]\boldsymbol{p}[c] = \sum_{f\in S_f} \boldsymbol{v}[f]\boldsymbol{p}[N_1(f)].$$

In the same way, we have

$$\sum_{c\in S_c}\sum_{f\in S_f} \boldsymbol{v}[f]\,\boldsymbol{p}[c]\,\mathbf{1}_{\{c=N_2(f)\}}\,\mathbf{1}_{\{f\in N(c)\}} = \sum_{f\in S_f} \boldsymbol{v}[f]\,\boldsymbol{p}[N_2(f)]$$

so that (5.54) yields

$$\sum_{c\in S_c} \mathtt{div}(\boldsymbol{v})[c]\,\boldsymbol{p}[c] + \sum_{f\in S_f} \mathtt{grad}(\boldsymbol{p})[f]\,\boldsymbol{v}[f] = 0. \qquad (5.55)$$

Until now, the boundary conditions have been ignored. They are included by introducing one more cell number $c = 0$ to denote the exterior. Then we can consider external faces and extend the mappings N_1 and N_2 to $S_c \cup \{0\}$ so that, if a given face f satisfies $N_1(f) = 0$ or $N_2(f) = 0$ then it is external. Note that the `grad` operator only defines values on internal faces. Now taking external faces into account, we obtain

$$\begin{aligned}&\sum_{c\in S_c} \mathtt{div}(\boldsymbol{v})[c]\,\boldsymbol{p}[c] + \sum_{f\in S_f} \mathtt{grad}(\boldsymbol{p})[f]\,\boldsymbol{v}[f]\\ &\qquad = \sum_{f\in \bar{S}_f\setminus S_f} \Big(\boldsymbol{p}[N_1(f)]\,\mathbf{1}_{\{N_2(f)=0\}} - \boldsymbol{p}[N_2(f)]\,\mathbf{1}_{\{N_1(f)=0\}}\Big)\,\boldsymbol{v}[f], \qquad (5.56)\end{aligned}$$

where $\bar{S}_f$ denotes the set of internal and external faces. The identity (5.56) is the discrete counterpart to

$$\int_\Omega p\nabla\cdot\vec{v}\,d\vec{x} + \int_\Omega \vec{v}\cdot\nabla p\,d\vec{x} = \int_{\partial\Omega} p\vec{v}\cdot\vec{n}\,ds. \tag{5.57}$$

Going back to (5.44), we see that the vector $\boldsymbol{v}\in\mathbb{R}^{n_f}$ is a discrete approximation of the flux on faces. Given $f\in S_f$, we have

$$\boldsymbol{v}[f] \approx \int_{\Gamma_f} \vec{v}(x)\cdot\vec{n}_f\,ds,$$

where $\vec{n}_f$ is the normal to the face f, where the orientation is given by the grid. The relation between the discrete pressure $\boldsymbol{p}\in\mathbb{R}^{n_c}$ and the discrete flux is given by the two-point flux approximation discussed in the previous section,

$$\boldsymbol{v}[f] = -\boldsymbol{T}[f]\,\mathtt{grad}(\boldsymbol{p})[f] \approx -\int_{\Gamma_f} \mathsf{K}(x)\nabla p\cdot\vec{n}_f\,ds, \tag{5.58}$$

where $\boldsymbol{T}[f]$ denotes the transmissibility of the face f, as defined in (5.49). Hence, the discretization of (5.44) is

$$\mathtt{div}(\boldsymbol{v}) = q \tag{5.59a}$$

$$v = -T\,\mathtt{grad}(\boldsymbol{p}). \tag{5.59b}$$

where the multiplication in (5.59b) holds element-wise.

5.4.3 Discretizing the time-of-flight and tracer equations

The transport equations (5.37) and (5.39) can be written on the common form

$$\nabla\cdot\big(u\vec{v}\big) = h(\vec{x},u), \tag{5.60}$$

where $u=\tau$ and $h=\phi+\tau\nabla\cdot\vec{v}$ for time-of-flight, and $u=C$ and $h=0$ for the stationary tracer distribution. To discretize the steady transport equation (5.60), we integrate it over a single grid cell Ω_i and use Gauss' divergence theorem to obtain

$$\int_{\partial\Omega_i} u\,\vec{v}\cdot\vec{n}\,ds = \int_{\Omega_i} h\big(\vec{x},u(\vec{x})\big)\,d\vec{x}.$$

In Section 5.4.1 we discussed how to discretize the flux over an interface Γ_{ik} between two cells Ω_i and Ω_k for the case that $u\equiv 1$. To be consistent with the notation used above, we will call this flux v_{ik}. If we can define an appropriate value u_{ik} at the interface Γ_{ik}, we can write the flux across the interface as

$$\int_{\Gamma_{ik}} u\,\vec{v}\cdot\vec{n}\,ds = u_{ik}v_{ik}. \tag{5.61}$$

The obvious idea of setting $u_{ik} = \frac{1}{2}(u_i + u_k)$ gives a centered scheme that is unfortunately notoriously unstable. By inspecting the direction information is propagating in the transport equation, we can instead use the so-called upwind value

$$u_{ik} = \begin{cases} u_i, & \text{if } v_{ij} \geq 0, \\ u_k, & \text{otherwise.} \end{cases} \tag{5.62}$$

This can be thought of as adding extra numerical dispersion which will stabilize the resulting scheme so that it does not introduce spurious oscillations.

For completeness, let us also write this discretization using the abstract notation defined in the previous section. If we discretize u by a vector $\boldsymbol{u} \in \mathbb{R}^{n_c}$ and h by a vector function $\boldsymbol{h}(\boldsymbol{u}) \in \mathbb{R}^{n_c}$, the transport equation (5.60) can be written in the discrete form

$$\mathtt{div}(\boldsymbol{uv}) = \boldsymbol{h}(\boldsymbol{u}). \tag{5.63}$$

We also substitute the expression for $\boldsymbol{v}$ from (5.59b) and use (5.62) to define u at each face f. Then, we define, for each face $f \in S_f$,

$$(\boldsymbol{uv})[f] = \boldsymbol{u}^f[f]\,\boldsymbol{T}[f]\,\mathtt{grad}(\boldsymbol{p})[f], \tag{5.64}$$

where

$$\boldsymbol{u}^f[f] = \begin{cases} \boldsymbol{u}[N_1(f)]), & \text{if } \mathtt{grad}(\boldsymbol{p})[f] > 0, \\ \boldsymbol{u}[N_2(f)]), & \text{otherwise.} \end{cases} \tag{5.65}$$

Time-of-flight and tracer distributions can of course also be computed based on tracing streamlines by solving the ordinary differential equations (5.33). The most commonly used method for tracing streamlines on hexahedral grids is a semi-analytical tracing algorithm introduced by Pollock [52], which uses analytical expressions of the streamline paths inside each cell based on the assumption that the velocity field is piecewise linear locally. Although Pollock's method is only correct for regular grids, it is often used also for highly skewed and irregular grids. Other approaches for tracing on unstructured grids and the associated accuracy are discussed in [20, 54, 34, 42, 32, 41, 37]. On unstructured polygonal grids, tracing of streamlines becomes significantly more involved. Because the general philosophy of MRST is that solvers should work independent of grid type, so that the user can seamlessly switch from structured to fully unstructured, polygonal grids, we prefer to use finite-volume methods rather than streamline tracing to compute time-of-flight and tracer distributions.

6

Incompressible Solvers in MRST

To form a full simulation model based upon the equations and discretizations introduced in the previous chapter, one generally needs the following components:

- A grid object, usually called `G`, describing the geometry of the reservoir.
- An object, usually called `rock`, that describes the petrophysical parameters of the reservoir.
- A fluid object describing fluid and rock-fluid properties.
- Objects to represent boundary conditions, source terms, and wells that may drive the flow in the reservoir.
- A state object holding the reservoir states (primary unknowns and derived quantities) like pressure, fluxes, face pressures, etc.

The implementation of grids and petrophysical properties in MRST was described in Chapters 3 and 2. In this chapter, we will present data structures and basic constructors used to represent fluid properties, reservoir states, and driving forces. Once these are in place, we move on to outline and discuss how the discretizations introduced in Section 5.4 have been implemented in `mrst-core` to provide a set of basic flow solvers for (single-phase) incomprssible flow. Then at the end, we will go through a few examples and give all code lines that are necessary for full simulation setups with various driving mechanisms.

6.1 Basic data structures

In this section we will outline the basic data structures that are needed to set up a single-phase simulation model.

6.1.1 Fluid properties

The only fluid properties we need in the basic single-phase flow equation are the viscosity and the fluid density for incompressible models and the fluid

compressibility for compressible models. For more complex single-phase and multiphase models, there are other fluid and rock-fluid properties that will be needed by flow and transport solvers. To simplify the communication of fluid properties, MRST uses so-called fluid object that contain basic fluid properties as well as a few function handles that can be used to evaluate rock-fluid properties that are relevant for multiphase flow. This basic structure can be expanded further to represent more advanced fluid models.

The following shows how to initialize a simple fluid object that only requires viscosity and density as input

```
fluid = initSingleFluid('mu' , 1*centi*poise, ...
                        'rho' , 1014*kilogram/meter^3);
```

After initialization, the fluid object will contain pointers to functions that can be used to evaluate petrophysical properties of the fluid:

```
fluid =
    properties: @(varargin)properties(opt,varargin{:})
    saturation: @(x,varargin)x.s
       relperm: @(s,varargin)relperm(s,opt,varargin{:})
```

Only the first function is relevant for single-phase flow, and returns the viscosity when called with a single output argument and the viscosity and the density when called with two output arguments. The other two functions can be considered as dummy functions that can be used to ensure that the single-phase fluid object is compatible with solvers written for more advanced fluid models. The `saturation` function accepts a reservoir state as argument (see Section 6.1.2) and returns the corresponding saturation (volume fraction of the fluid phase) which will either be empty or set to unity, depending upon how the reservoir state has been initialized. The `relperm` function accepts a fluid saturation as argument and returns the relative permeability, i.e., the reduction in permeability due to the presence of other fluid phases, which is always identical to one for a single-phase model.

6.1.2 Reservoir states

To hold the dynamic state of the reservoir, MRST uses a special date structure. We will in the following refer to realizations of this structure as state objects. In its basic form, the structure contains three elements: a vector `pressure` with one pressure per cell in the model, a vector `flux` with one flux per grid face in the model, and a vector `s` with one saturation value for each cell, which should either be empty or be an identity vector since we only have a single fluid. The state object is typically initialized by a call to the following function

```
state = initResSol(G, p0, s0);
```

where `p0` is the initial pressure and `s0` is an optional parameter that gives the initial saturation (which should be identical to one for single-phase models).

Notice that this initialization *does not* initialize the fluid pressure to be at hydrostatic equilibrium. If such a condition is needed, it must be enforced explicitly by the user. In the case that the reservoir has wells, one should use the alternative function:

```
state = initState(G, W, p0, s0);
```

This will give a state object with an additional field `wellSol`, which is a vector with length equal the number of wells. Each element in the vector is a structure that has two fields `wellSol.pressure` and `wellSol.flux`. These two fields are vectors of length equal the number of completions in the well and contain the bottom-hole pressure and flux for each completion.

6.1.3 Fluid sources

The simplest way to describe flow into or flow out from interior points of the reservoir is to use volumetric source terms. These source terms can be added using the following function:

```
src = addSource(src, cells, rates);
src = addSource(src, cells, rates, 'sat', sat);
```

Here, the input values are:

- `src`: structure from a prior call to `addSource` which will be updated on output or an empty array (`src==[]`) in which case a new structure is created. The structure contains the following fields:
  - `cell`: cells for which explicit sources are provided
  - `rate`: rates for these explicit sources
  - `value`: pressure or flux value for the given condition
  - `sat`: fluid composition of injected fluids in cells with rate>0
- `cells`: indices to the cells in the grid model in which this source term should be applied.
- `rates`: vector of volumetric flow rates, one scalar value for each cell in `cells`. Note that these values are interpreted as flux rates (typically in units of [m^3/day] rather than as flux density rates (which must be integrated over the cell volumes to obtain flux rates).
- `sat`: optional parameter that specifies the composition of the fluid injected from this source. An $n \times m$ array of fluid compositions with n being the number of elements in `cells` and m is the number of fluid phases. For $m = 3$, the columns are interpreted as: 1='aqua', 2='liquid', and 3='vapor'. This field is for the benfit of multiphase transport solvers, and is ignored for all sinks (at which fluids flow *out* of the reservoir). The default value is `sat` = [], which corresponds to single-phase flow. As a special case, if `size(sat,1)==1`, then the saturation value will be repeated for all cells specified by `cells`.

For convenience, values and sat may contain a single value; this value is then used for all faces specified in the call.

There can only be a single net source term per cell in the grid. Moreover, for incompressible flow with no-flow boundary conditions, the source terms *must* sum to zero if the model is to be well posed, or alternatively sum to the flux across the boundary. If not, we would either inject more fluids than we extract, or vice versa, and hence implicitly violate the assumption of incompressbilitity.

6.1.4 Boundary conditions

As discussed in Section 5.3.1, all outer faces in a grid model are assumed to be no-flow boundaries in MRST unless other conditions are specified explicitly. The basic mechanism for specifying Dirichlet and Neumann boundary conditions is to use the function:

```
bc = addBC(bc, faces, type, values);
bc = addBC(bc, faces, type, values, 'sat', sat);
```

Here, the input values are:

- `bc`: structure from a prior call to `addBC` which will be updated on output or an empty array (`bc==[]`) in which case a new structure is created. The structure contains the following fields:
  - `face`: external faces for which explicit conditions are set
  - `type`: cell array of strings denoting type of condition
  - `value`: pressure or flux value for the given condition
  - `sat`: fluid composition of fluids passing through inflow faces, not used for single-phase models
- `faces`: array of external faces at which this boundary condition is applied.
- `type`: type of boundary condition. Supported values are 'pressure' and 'flux', or cell array of such strings.
- `values`: vector of boundary conditions, one scalar value for each face in `faces`. Interpreted as a pressure value in units of [Pa] when `type` equals 'pressure' and as a flux value in units of [m^3/s] when `type` is 'flux'. If the latter case, the positive values in `values` are interpreted as injection fluxes *into* the reservoir, while negative values signify extraction fluxs, i.e., fluxes *out of* the reservoir.
- `sat`: optional parameter that specifies the composition of the fluid injected across inflow faces. Similar setup as for explained for source terms in Section 6.1.3.

There can only be a single boundary condition per face in the grid. Solvers assume boundary conditions are given on the boundary; conditions in the interior of the domain yield unpredictable results. Moreover, for incompressible flow and only Neumann conditions, the boundary fluxes *must* sum to zero if the model is to be well posed. If not, we would either inject more fluids

than we extract, or vice versa, and hence implicitly violate the assumption of incompressbilitity.

For convenience, MRST also offers two additional routines that can be used to set Dirichlet and Neumann conditions at all outer faces in a certain direction for grids having a logical IJK numbering:

```
bc = pside(bc, G, side, p);
bc = fluxside(bc, G, side, flux)
```

The `side` argument is a string that much match one out of the following six alias groups:

```
1: 'West',   'XMin',  'Left'
2: 'East',   'XMax',  'Right'
3: 'South',  'YMin',  'Back'
4: 'North',  'YMax',  'Front'
5: 'Upper',  'ZMin',  'Top'
6: 'Lower',  'ZMax',  'Bottom'
```

These groups correspond to the cardinal directions mentioned as the first alternative in each group. The user should also be aware of an important difference in how fluxes are specified in `addBC` and `fluxside`. Specifying a scalar value in `addBC` means that this value will be copied to all faces the boundary condition is applied to, whereas a scalar value in `fluxside` sets the cummulative flux for all faces that make up the global side to be equal the specified value.

6.1.5 Wells

Wells are similar to source terms in the sense that they describe injection or extraction of fluids from the reservoir, but differ in the sense that they not only provide a volumetric flux rate, but also contain a model that couples this flux rate to the difference between the average reservoir in the grid cell and the pressure inside the wellbore. As discussed in Section 5.3.2, this relation can be written for each perforation as

$$v_p = \frac{\mathrm{WI}}{\mu}(p_i - p_f) \tag{6.1}$$

where WI is the well index, p_i is the pressure in the perforated grid cell, and p_f is the flowing pressure in the wellbore. The latter can be found from the pressure at the top of hte well and the density of the fluid in each perforation. For single-phase, incompressible this $p_f = p_{wh} + \rho \Delta z_f$, where p_{wh} is the pressure at the well head and Δz_f is the vertical distance from this point and to the perforation.

In MRST, the structure used to represent wells, which by convention is typically called `W`, consists of the following fields:

- `cells`: an array index to cells perforated by this well
- `type`: string describing which variable is controlled (i.e., assumed to be fixed), either 'bhp or 'rate'
- `val`: the target value of the well control (pressure value for `type`='bhp' or the rate for `type`='rate'.
- `r`: the wellbore radius (double).
- `dir`: a char describing the direction of the perforation, one of the cardinal directions 'x', 'y' or 'z'
- `WI`: the well index: either the productivity index or the well injectivity index depending on whether the well is producing or injecting.
- `dZ`: the height differences from the well head, which is defined as the 'highest' contact (i.e., the contact with the minimum z-value counted amongst all cells perforated by this well)
- `name`: string giving the name of the well
- `compi`: fluid composition, only used for injectors
- `refDepth`: reference depth of control mode
- `sign`: define if the well is intended to be producer or injector

Well structures are created by a call to the function

```
W = addWell(W, G, rock, cellInx);
W = addWell(W, G, rock, cellInx, 'pn', pv, ... );
```

Here, `cellInx` is a vector of indices to the cells perforated by the well, and 'pn'/pv denote one or more 'key'/value pairs that can be used to specify optional parameters that influence the well model:

- `type`: string specifying well control, 'bhp' (default) means that the well is controlled by bottom-hole pressure, whereas 'rate' means that the well is rate controlled.
- `val`: target for well control. Interpretation of this values depends upon `type`. For 'bhp' the value is assumed to be in unit Pascal, and for 'rate' the value is given in unit [m^3/sec]. Default value is 0.
- `radius`: wellbore radius in meters. Either a single, scalar value that applies to all perforations, or a vector of radii, with one value for each perforation. The default radius is 0.1 m.
- `dir`: well direction. A single CHAR applies to all perforations, while a CHAR array defines the direction of the corresponding perforation.
- `innerProduct`: used for consistent discretizations discussed in Chapter 8
- `WI`: well index. Vector of length equal the number of perforations in the well. The default value is -1 in all perforations, whence the well index will be computed from available data (cell geometry, petrophysical data, etc) in grid cells containing well completions
- `Kh`: permeability times thickness. Vector of length equal the number of perforations in the well. The default value is -1 in all perforations, whence the thickness will be computed from the geometry of each perforated cell.

– `skin`: skin factor for computing effective well bore radius. Scalar value or vector with one value per perforation. Default value: 0.0 (no skin effect).
– `Comp_i`: fluid composition for injection wells. Vector of saturations. Default value: `Comp_i` = [1, 0, 0] (water injection)
– `Sign`: well type: production (`sign`=−1) or injection (`sign`=1). Default value: [] (no type specified)
– `name`: string giving the name of the well. Default value is 'Wn' where n is the number of this well, i.e., `n`=`numel(W)`+1

For convenience, MRST also provides the function

```
W = verticalWell(W, G, rock, I, J, K)
W = verticalWell(W, G, rock, I,    K)
```

that can be used to specify vertical wells in models described by Cartesian grids or grids that have some kind of extruded structure. Here,

– `I,J`: gives the horizontal location of the well heel. In the first mode, both `I` and `J` are given and then signify logically Cartesian indices so that `I` is the index along the first logical direction while `J` is the index along the second logical direction. This mode is only supported in grids which have an underlying Cartesian (logical) structure such as purely Cartesian grids or corner-point grids.
  In the second mode, only `I` is described and gives the *cell index* of the topmost cell in the column through which the vertical well will be completed. This mode is supported for logically Cartesian grids containing a three-component field `G.cartDims` or for otherwise layered grids which contain the fields `G.numLayers` and `G.layerSize`.
– `K`: a vector of layers in which this well should be completed. If `isemmpty(K)` is true, then the well is assumed to be completed in all layers in this grid column and the vector is replaced by 1:`num_layers`.

6.2 Incompressible two-point pressure solver

In MRST, the two-point flux-approximation scheme introduced in Section 5.4.1 is implemented as two different routines:

```
hT = computeTrans(G,rock)
```

computes the half-face transmissibilities and does not depend on the fluid model, the reservoir state, or the driving mechanisms, whereas

```
state = incompTPFA(state, G, hT, fluid, 'mech1', obj1, ...)
```

takes the complete model description as input and assembles and solves the two-point system. Here, `mech` arguments the drive mechanism ('src',

'bc', and/or 'wells') using correctly defined objects `obj`, as discussed in Sections 6.1.3–6.1.5. Notice that `computeTrans` may fail to compute sensible transmissibilities of the permeability field in `rock` is not given in SI units. Likewise, `incompTPFA` may produce strange results if the inflow and outflow specified by the boundary conditions, source terms, and wells does not sum to zero and hence violates the assumption of incompressibility. However, if fixed pressure is specified in wells or on parts of the outer boundary, there will be an outflow or inflow that will balance the net rate that is specified elsewhere. In the remains of this section, we will discuss more details of the incompressible solver and demonstrate how simple it is to implement the TPFA method on general polyhedral grid by going through the essential code lines needed to compute half-transmissibilities and solve and assemble the global system. The impatient reader can jump directly to Section 6.4, in which we go through several examples that demonstrate the use of the incompressible solver for single-phase flow.

To focus on the discretization and keep the discussion simple, we will not look at the full implementation of the two-point solver in `mrst-core`. Instead, we will discuss excerpts from two simplified functions, `simpleComputeTrans` and `simpleIncompTPFA`, that are located in the `1phase` directory of the `mrst-book` module and together form a simplified single-phase solver that has been created for pedagogical purposes. The standard `computeTrans` function can be used for different representations of petrophysical parameters and includes functionality to modify the discretization by overriding the definition of cell and face centers and/or including multipliers that modify the values of the half-transmissibilities, see e.g., Sections 2.3.3 and 2.4.5. Likewise, the `incompTPFA` solver is implemented for a general, incompressible flow model with multiple fluid phases with flow driven by a general combination of boundary conditions, fluid sources, and well models.

Assuming that we have a standard grid `G` that contains cell and face centroids, e.g., as computed by `computeGeometry` as discussed in Section 3.4, the essential code lines of `simpleComputeTrans` are as follows: First, we define the vectors $\vec{c}_{i,k}$ from cell centroids to face centroids. To this end, we first need to determine the map from faces to cell number so that the correct cell centroid is subtracted from each face centroid.

```
hf2cn = gridCellNo(G);
C = G.faces.centroids(G.cells.faces(:,1),:) - G.cells.centroids(hf2cn,:);
```

The face normals in MRST are assumed to have length equal to the corresponding face areas, and hence correspond to $A_{i,k}\vec{n}_{i,k}$ in (5.48). To get the correct sign, we look at the neighboring information that describes which cells share the face: if the current cell number is in the first column, the face normal has a positive sign. If not, it gets a negative sign:

```
sgn = 2*(hf2cn == G.faces.neighbors(G.cells.faces(:,1), 1)) - 1;
N   = bsxfun(@times, sgn, G.faces.normals(G.cells.faces(:,1),:));
```

The permeability tensor may be stored in different formats, as discussed in Section 2.4, and we therefore use an utility function to extract it:

```
[K, i, j] = permTensor(rock, G.griddim);
```

Finally, we compute the half transmissibilities, $C^T K N / C^T C$. To limit memory use, this is done in a for-loop (which are rarely used in MRST):

```
hT = zeros(size(hf2cn));
for k=1:size(i,2),
   hT = hT + C(:,i(k)) .* K(hf2cn, k) .* N(:, j(k));
end
hT = hT./ sum(C.*C,2);
```

The actual code has a few additional lines the perform various safeguards and consistency checks.

Once the half transmissibilities have been computed, they can be passed to the `simpleIncompTPFA` solver. The first thing this solver needs to do is adjust the half transmissibilities to account for fluid viscosity, since they were derived for a fluid with unit viscosity:

```
mob = 1./fluid.properties(state);
hT  = hT .* mob(hf2cn);
```

Then we loop through all faces and compute the face transmissibility as the harmonic average of the half-transmissibilities

```
T = 1 ./ accumarray(G.cells.faces(:,1), 1 ./ hT, [G.faces.num, 1]);
```

Here, we have used the MATLAB function `accumarray` which constructs an array by accumulation. A call to `a = accumarray(subs,val)` will use the subscripts in `subs` to create an array `a` based on the values `val`. Each element in `val` has a corresponding row in `subs`. The function collects all elements that correspond to identical subscripts in `subs` and stores the sum of those values in the element of `a` corresponding to the subscript. In our case, `G.cells.faces(:,1)` gives the global face number for each half face, and hence the call to `accumarray` will sum the transmissibilities of the half-faces that correspond to a given global face and store the result in the correct place in a vector of `G.faces.num` elements. The function `accumarray` is a very powerful function that is used a lot in MRST in place of nested for-loops. In fact, we will use this function to loop over all the cells in the grid and collect and sum the transmissibilities of the faces of each cell to define the diagonal of the TPFA matrix:

```
nc = G.cells.num;
i  = all(G.faces.neighbors ~= 0, 2);
d  = accumarray(reshape(G.faces.neighbors(i,:), [], 1), ...
                repmat(T(i), [2,1]),    [nc, 1]);
```

Now that we have computed both the diagonal and the off-diagonal element of A, the discretization matrix itself can be constructed by a straightforward call to MATLAB's `sparse` function:

```
I  = [G.faces.neighbors(i,1); G.faces.neighbors(i,2); (1:nc)'];
J  = [G.faces.neighbors(i,2); G.faces.neighbors(i,1); (1:nc)'];
V  = [-T(i); -T(i); d]; clear d;
A  = sparse(double(I), double(J), V, nc, nc);
```

Finally, we check if Dirichlet boundary conditions are imposed on the system, and if not we modify the first element of the system matrix to (somewhat arbitrarily) fix the pressure in the first cell to zero, before we solve the system to compute the pressure:

```
A(1) = 2*A(1);
p = mldivide(A, rhs);
```

As linear solver we have used MATLAB's default solver `mldivide`, which for a sparse system boils down to calling a direct solver from UMFPACK that is based on a unsymmetric, sparse, multifrontal LU factorization add citation. While this solver is efficient for small to medium-sized systems, larger systems are more efficiently solved using more problem-specific solvers. To provide flexibility, the linear solver can be passed as a function-pointer argument to both `incompTPFA` and `simpleIncompTPFA`.

Once the pressures have been computed, we can compute pressure values at the face centroids using the half-face transmissibilities

```
fp = accumarray(G.cells.faces(:,1), p(hf2cn).*hT, [G.faces.num,1])./ ...
     accumarray(G.cells.faces(:,1), hT, [G.faces.num,1]);
```

and likewise construct the fluxes across the interior faces

```
ni   = G.faces.neighbors(i,:);
flux = -accumarray(find(i), T(i).*(p(ni(:,2))-p(ni(:,1))), [nf, 1]);
```

In the code excerpts given above, we did not account for gravity forces and boundary conditions, which both will complicate the code beyond the scope of the current presentation. The interested reader should consult the actual code to work out these details.

6.3 Upwind solver for time-of-flight and tracer

In `mrst-core` the upwind, finite-volume discretization introduced in Section 5.4.3 is implemented in the function

```
tof = computeTimeOfFlight(state, G, rock, mech1, obj1, ...)
```

whose main purpose is to solve the time-of-flight equation. The `mech` arguments specify the drive mechanism ('src', 'bc', and/or 'wells') specified in terms of specific objects `obj`, as discussed in Sections 6.1.3 to 6.1.5. Tracer partitions can also be computed if the user specifies extra input parameters. In the following, we will go through the main parts of how this discretization is implemented.

We start by identifying all volumetric sources of inflow and outflow, and collect the results in a vector `q` of source terms having one value per cell

```
q = sparse(src.cell, 1, src.rate, G.cells.num, 1);
```

We also need to compute the accumulated inflow and outflow from boundary fluxes for each cell. This will be done in three steps. First, we create an empty vector `ff` with one entry per global face, find all faces that have Neumann conditions, and insert the corresponding value in the correct row

```
ff    = zeros(G.faces.num, 1);
isNeu = strcmp('flux', bc.type);
ff(bc.face(isNeu)) = bc.value(isNeu);
```

For faces having Dirichlet boundary conditions, the flux is not specified and must be extracted from the solution computed by the pressure solver, i.e., from the `state` object that holds the reservoir state. We also need to set the correct sign so that fluxes *into* a cell are positive and fluxes *out of* a cell are negative. To this end, we use the fact that the normal vector of face `i` points from cell `G.faces.neighbors(i,1)` to `G.faces.neighbors(i,2)`. In other words, the sign of the flux across an outer face is correct if `neighbors(i,1)==0`, but if `neighbors(i,2)==0` we need to reverse the sign

```
isDir = strcmp('pressure', bc.type);
i     = bc.face(isDir);
if ~isempty(i)
   ff(i) = state.flux(i) .* (2*(G.faces.neighbors(i,1)==0) - 1);
end
```

The last step is to sum all the fluxes across outer faces and collect the result in a vector `qb` that has one value per cell

```
is_outer = ~all(double(G.faces.neighbors) > 0, 2);
qb = sparse(sum(G.faces.neighbors(is_outer,:), 2), 1, ...
            ff(is_outer), G.cells.num, 1);
```

Here, `G.faces.neighbors(is_outer,:), 2)` gives the index of the cell that is attached to each outer face (since the entry in one of the columns must be zero for an outer face).

Once the contributions to inflow and outflow are collected, we build an upwind flux matrix A defined such that $A_{ji} = \max(v_{ij}, 0)$ and $A_{ij} = -\min(v_{ij}, 0)$, where v_{ij} is the flux computed by the TPFA scheme discussed in the previous section.

```
i  = ~any(G.faces.neighbors==0, 2);
n  = double(G.faces.neighbors(i,:));
nc = G.cells.num;
A  = sparse(n(:,2), n (:,1),   max(state.flux(i), 0), nc, nc) ...
   + sparse(n(:,1), n (:,2),  -min(state.flux(i), 0), nc, nc);
```

Then the diagonal of the discretization matrix is obtained by summing rows in the upwind flux matrix. This will give the correct diagonal in all cell except for those with a positive fluid source. In these cells, we can as a reasonable approximation set the average time-of-flight to be equal half the time it takes to fill the cell, which means that the diagonal entry should be equal twice the fluid rate inside the cell:

```
A  = -A + spdiags(sum(A,2)+2*max(q+qb,0), 0, nc, nc);
```

Finally, we subtract the divergence of the velocity minus any source terms from the diagonal of A to account for compressibility effects.

```
div = accumarray(gridCellNo(G), faceFlux2cellFlux(G, state.flux));
A   = A - spdiags(div-q, 0, nc, nc);
```

We have now established the complete discretization matrix, and time-of-flight can be computed by a simple matrix inversion

```
tof  = A \ poreVolume(G,rock);
```

If there are no gravity forces and the flux has been computed by a monotone scheme, one can show that the discretization matrix A can be permuted to a lower-triangular form [45, 44]. In the general case, the permuted matrix will be block triangular with irreducible diagonal blocks. Such systems can be inverted very efficiently using a permuted backsubstitution algorithm as long as the irreducible diagonal blocks are small. In our experience, MATLAB is quite good at detecting such structures and using the simple backslash (\) operator is therefore efficient, even for quite large models.

In addition to time-of-flight, we can compute stationary tracers as discussed in Section 5.3.4. This is done by passing an optional parameter,

```
tof = computeTimeOfFlight(state, G, rock, ..., 'tracer', tr)
```

where `tr` is a cell-array of vectors that each give the indexes of the cells that will be assigned a unique color. For incompressible flow, the discretization matrix of the tracer equation is the same as that for time-of-flight, and all we need to do is to assemble the right-hand side

```
numTrRHS = numel(tr);
TrRHS = zeros(nc,numTrRHS);
for i=1:numTrRHS,
   TrRHS(tr{i},i) = 2*qp(tr{i});
end
```

Since we have doubled the rate in any cells with a positive source when constructing the matrix A, we need to also double the rate on the right-hand side.

Now we can solve the combined time-of-flight, tracer problem as a linear system with multiple right-hand side,

```
T  = A \ [poreVolume(G,rock) TrRHS];
```

which means that we essentially get the tracer for free as long as the number of tracers does not exceed MATLAB's limit for the number of right-hand columns that can be handled in one solve.

6.4 Simulation examples

You have now been introduced to all the functionality from `mrst-core` that is necessary to solve a single-phase flow problem. In following, we will discuss several examples, in which we demonstrate step-by-step, how to set up flow model, solve them, and visualize and analyze the resulting flow fields. Complete codes can be found in the `1phase` directory of the `mrst-book` module.

6.4.1 Quarter-five spot

As our first example, we show how to solve (5.44) with no-flow boundary conditions and two source terms at diagonally opposite corners of a 2D Cartesian grid covering a 500×500 m^2 area. This setup mimics a quarter five-spot well pattern, which is a standard test in reservoir simulation. The full code is available in the script `quarterFiveSpot.m`.

We use a $n_x \times n_y$ grid with homogeneous petrophysical data, permeability of 100 mD and porosity of 0.2:

```
[nx,ny] = deal(32);
G = cartGrid([nx,ny],[500,500]);
G = computeGeometry(G);
rock.perm = ones(G.cells.num, 1)*100*milli*darcy;
rock.poro = ones(G.cells.num, 1)*.2;
```

As we saw above, all we need to know to develop the spatial discretization is the reservoir geometry and the petrophysical properties. This means that we can compute the half transmissibilities without knowing any details about the fluid properties and the boundary conditions and/or sources/sinks that will drive the global flow:

```
hT = simpleComputeTrans(G, rock);
```

The result of this computation is a vector with one value per local face of each cell in the grid, i.e., a vector with `G.cells.faces` entries.

The reservoir is horizontal and gravity forces are therefore not active. We create a fluid with properties that are typical for water:

```
gravity reset off
fluid = initSingleFluid('mu' , 1*centi*poise, ...
                        'rho', 1014*kilogram/meter^3);
```

To drive the flow, we will use a fluid source at the south-west corner and a fluid sink at the north-east corner of the model. The time scale of the problem is defined by the strength of the source term. In our case, we set the source terms such that a unit time corresponds to the injection of one pore volume of fluids. All flow solvers in MRST automatically assume no-flow conditions on all outer (and inner) boundaries if no other conditions are specified explicitly.

```
pv  = sum(poreVolume(G,rock));
src = addSource([], 1, pv);
src = addSource(src, G.cells.num, -pv);
display(src)
```

The data structure used to represent the fluid sources contains three elements: a vector `src.cell` of cell numbers, a vector `src.rate` with the fluid rate (positive for inflow into the reservoir and negative for outflow from the reservoir), and a vector `src.sat` with the fluid saturation (which has no meaning here and is hence set to be empty)

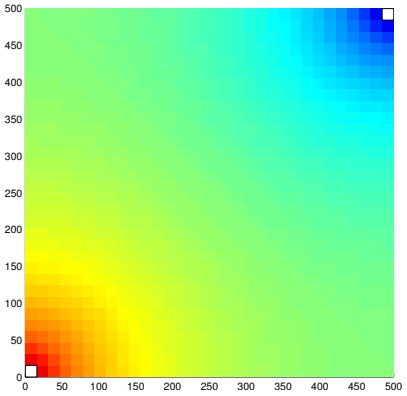
```
src =
    cell: [2x1 double]
    rate: [2x1 double]
     sat: []
```

To simplify communication among different flow and transport solvers, all unknowns (reservoir states) are collected in a structure. Strictly speaking, this structure need not be initialized for an incompressible model in which none of the fluid properties depend on the reservoir states. However, to avoid treatment of special cases, MRST requires that the structure is initialized and passed as argument to the pressure solver. We therefore initialize it with a dummy pressure value of zero and a unit fluid saturation since we only have a single fluid

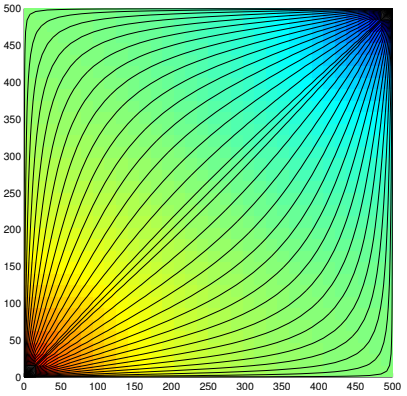
```
state = initResSol(G, 0.0, 1.0);
display(state)
```

```
state =
    pressure: [1024x1 double]
        flux: [2112x1 double]
           s: [1024x1 double]
```

This completes the setup of the model. To solve for the pressure, we simply pass the reservoir state, grid model, half transmissibilities, fluid model, and




```
plotCellData(G, state.pressure);
plotGrid(G, src.cell, 'FaceColor', 'w');
axis equal tight; colormap(jet(128));
```




```
hf = streamline(Sf);
hb = streamline(Sb);
set([hf; hb], 'Color','k');
```

Fig. 6.1. Solution of the quarter five-spot problem on a 32×32 uniform grid. The left plot shows the pressure distribution and in the right plot we have imposed streamlines passing through centers of the cells on the NW–SE diagonal.

driving forces to the flow solver that assembles and solves the incompressible flow equation.

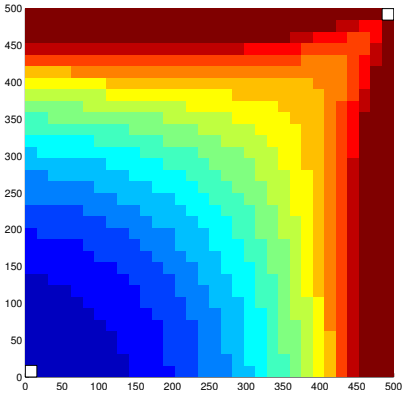
```
state = simpleIncompTPFA(state, G, hT, fluid, 'src', src);
display(state)
```

As explained above, `simpleIncompTPFA` solves for pressure as the primary variable and then uses transmissibilities to reconstruct the face pressure and intercell fluxes. After a call to the pressure solver, the `state` object is therefore expanded by a new field `facePressure` that contains pressures reconstructed at the face centroids

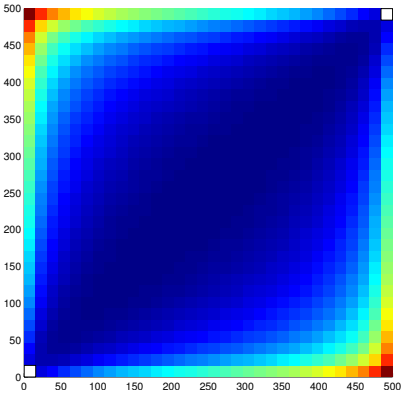
```
state =
        pressure: [1024x1 double]
            flux: [2112x1 double]
               s: [1024x1 double]
    facePressure: [2112x1 double]
```

Figure 6.1 shows the resulting pressure distribution. To improve the visualization of the flow field, we show streamlines. In MRST, Pollock's method [52] for semi-analytical tracing of streamlines has been implemented in the `streamlines` add-on module. Here, we will use the method to trace streamlines forward and backward, starting from the midpoint of all cells along the NW–SE diagonal in the grid

```
mrstModule add streamlines;
seed = (nx:nx-1:nx*ny).';
Sf = pollock(G, state, seed, 'substeps', 1);
Sb = pollock(G, state, seed, 'substeps', 1, 'reverse', true);
```




```
plotCellData(G, toff);
plotGrid(G, src.cell, 'FaceColor','w');
axis equal tight;
colormap(jet(16)); caxis([0,1]);
```




```
plotCellData(G, toff+tofb);
plotGrid(G, src.cell, 'FaceColor','w');
axis equal tight;
colormap(jet(128));
```

Fig. 6.2. Solution of the quarter five-spot problem on a 32×32 uniform grid. The left plot shows time-of-flight plotted with a few color levels to create a crude contouring effect. The right plot shows a plot of the total travel time to distinguish high-flow and stagnant regions.

The `pollock` routine produces a cell array of individual streamlines that can be passed onto MATLAB's built-in `streamline` routine for plotting, as shown to the right in Figure 6.1.

To get a better picture of how fast the fluids will flow through our domain, we solve the time-of-flight equation (5.37) subject to the condition that $\tau = 0$ at the inflow, i.e., at all points where $q > 0$. For this purpose, we use the `computeTimeOfFlight` solver discussed in Section 6.3, which can compute both the forward time-of-flight from inflow points and into the reservoir,

```
toff = computeTimeOfFlight(state, G, rock, 'src', src);
```

and the backward time-of-flight from outflow points and backwards into the reservoir

```
tofb = computeTimeOfFlight(state, G, rock, 'src', src, 'reverse', true);
```

Isocontours of time-of-flight define natural time lines in the reservoir, and to emphasize this fact, the left plot in Figure 6.2 shows the time-of-flight plotted using only a few colors to make a rough contouring effect. The sum of the forward and backward time-of-flights give the total time it takes for a fluid particle to travel through the reservoir, from an inflow point to an outflow point. The total travel time can be used to visualize high-flow and stagnant regions as demonstrated in the right plot of Figure 6.2.

COMPUTER EXERCISES:

Rerun the quarter five-spot example with the following modifications:

- Replace the Cartesian grid by a curvilinear grid, e.g., using `twister` or a random perturbation of internal nodes as shown in Figure 3.3.
- Replace the homogeneous permeability by a heterogeneous permeability derived from the Carman–Kozeny relation (2.5)
- Set the domain to be a single layer of the SPE10 model. Hint: use `SPE10_rock()` to sample the petrophysical parameters and remember to convert to SI units.

6.4.2 Boundary conditions

To demonstrate how to specify boundary conditions, we will go through essential code lines of three different examples; the complete scripts can be found in `boundaryConditions.m`. In all three examples, the reservoir is 50 meter thick, is located at a depth of approximately 500 meters, and is restricted to a 1×1 km^2 area. The permeability is uniform and anisotropic, with a diagonal (1000, 300, 10) mD tensor, and the porosity is uniform and equal 0.2 In the first two examples, the reservoir is represented as a $20 \times 20 \times 5$ rectangular grid, and in the third example the reservoir is given as a corner-point grid of the same Cartesian dimension, but with an uneven uplift and four intersecting faults (as shown in the left plot of Figure 3.31):

```
[nx,ny,nz] = deal(20, 20, 5);
[Lx,Ly,Lz] = deal(1000, 1000, 50);
switch setup
   case 1,
      G = cartGrid([nx ny nz], [Lx Ly Lz]);
   case 2,
      G = cartGrid([nx ny nz], [Lx Ly Lz]);
   case 3,
      G = processGRDECL(makeModel3([nx ny nz], [Lx Ly Lz/5]));
      G.nodes.coords(:,3) = 5*(G.nodes.coords(:,3) ...
                              - min(G.nodes.coords(:,3)));
end
G.nodes.coords(:,3) = G.nodes.coords(:,3) + 500;
```

Setting rock and fluid parameters, computing transmissibilities, and initializing the reservoir state can be done as explained in the previous section, and details are not included for brevity.

Linear pressure drop

In the first example (`setup`=1), we specify a Neumann condition with total inflow of 5000 m^3/day on the east boundary and a Dirichlet condition with fixed pressure of 50 bar on the west boundary:

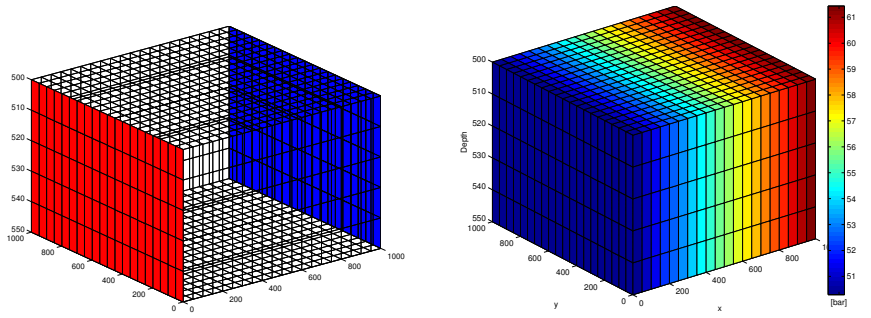


Fig. 6.3. First example of a flow driven by boundary conditions. In the left plot, faces with Neumann conditions are marked in blue and faces with Dirichlet conditions are marked in red. The right plot shows the resulting pressure distribution.

```
bc = fluxside(bc, G, 'EAST', 5e3*meter^3/day);
bc = pside  (bc, G, 'WEST', 50*barsa);
```

This completes the definition of the model, and we can pass the resulting objects to the `simpleIncompTFPA` solver to compute the pressure distribution shown to the right in Figure 6.3. In the absence of gravity, these boundary conditions will result in a linear pressure drop from east to west inside the reservoir.

Hydrostatic boundary conditions

In the next example, we will use the same model, except that we now include the effects of gravity and assume hydrostatic equilibrium at the outer vertical boundaries of the model. First, we initialize the reservoir state according to hydrostatic equilibrium, which is straightforward to compute if we for simplicity assume that the overburden pressure is caused by a column of fluids with the exact same density as in the reservoir:

```
state = initResSol(G, G.cells.centroids(:,3)*rho*norm(gravity), 1.0);
```

There are at least two different ways to specify hydrostatic boundary conditions. The simplest approach is to use the function `psideh`, i.e.,

```
bc = psideh([], G,  'EAST', fluid);
bc = psideh(bc, G, 'WEST', fluid);
bc = psideh(bc, G, 'SOUTH', fluid);
bc = psideh(bc, G, 'NORTH', fluid);
```

Alternatively, we can do it manually ourselves. To this end, we need to extract the reservoir perimeter defined as all exterior faces are vertical, i.e., whose normal vector have no z-component,

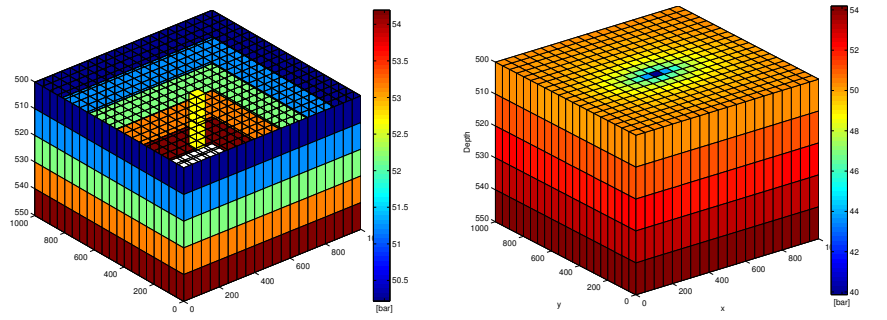


Fig. 6.4. A reservoir with hydrostatic boundary condition and fluid extracted from a sink penetrating two cells in the upper two layers of the model. The left plot shows the boundary and the fluid sink, while the right plot shows the resulting pressure distribution.

```
f = boundaryFaces(G);
f = f(abs(G.faces.normals(f,3))<eps);
```

To get the hydrostatic pressure at each face, we can either compute it directly by using the face centroids,

```
fp = G.faces.centroids(f,3)*rho*norm(gravity);
```

or we use the initial equilibrium that has already been established in the reservoir by can sample from the cells adjacent to the boundary

```
cif = sum(G.faces.neighbors(f,:),2);
fp  = state.pressure(cif);
```

The latter may be useful if the initial pressure distribution has been computed by a more elaborate procedure than what is currently implemented in `psideh`. In either case, the boundary conditions can now be set by the call

```
bc = addBC(bc, f, 'pressure', fp);
```

To make the problem a bit more interesting, we also include a fluid sink at the midpoint of the upper two layers in the model,

```
ci = round(.5*(nx*ny-nx));
ci = [ci; ci+nx*ny];
src = addSource(src, ci, repmat(-1e3*meter^3/day,numel(ci),1));
```

The boundary conditions and source terms are shown to the left in Figure 6.4 and the resulting pressure distribution to the right. The fluid sink will cause a pressure draw-down, which will have an ellipsoidal shape because of the anisotropy in the permeability field.

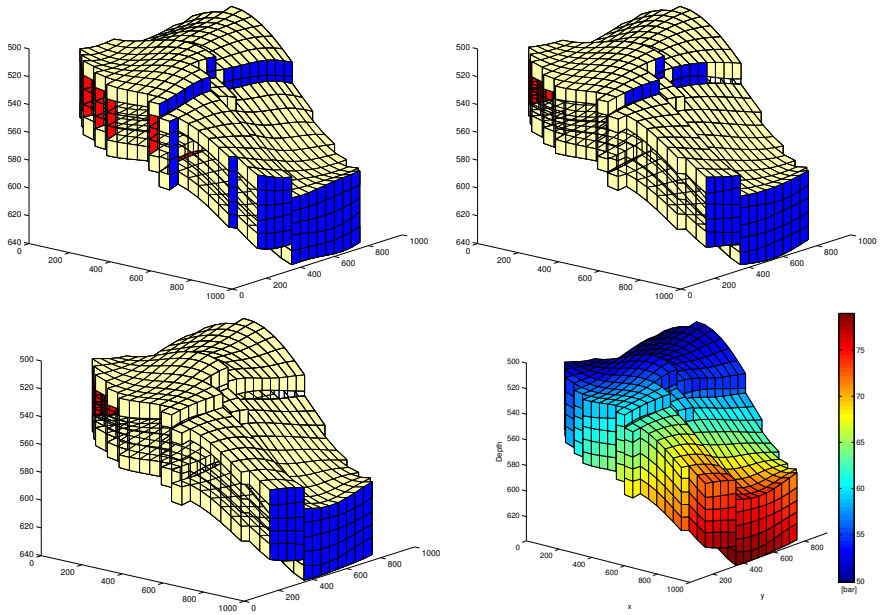


Fig. 6.5. Specifying boundary conditions along the outer perimeter of a corner-point model. The upper-left plot shows the use of `fluxside` (blue color) and `pside` (red color) to set boundary conditions on the east and west global boundaries. In the upper-right point, the same functions have been used along with a specification of subranges in the global sides. In the lower-left plot, we have utilized user-supplied information to correctly set the conditions only along the perimeter. The lower-right plot shows the resulting pressure solution.

Conditions on non-rectangular domain

In the last example, we consider a case where the outer boundary of the reservoir is not a simple hexahedron. In such cases, it may not be as simple as above to determine the exterior faces that lie on the perimeter of the reservoir. In particular, faults having a displacement may give exterior faces at the top an bottom of the model that are not part of what one would call the reservoir perimeter when setting boundary conditions other than no-flow. Likewise, other geological processes like erosion may cause gaps in the model that lead to exterior faces that are not part of the natural perimeter. This is illustrated in the left plot of Figure 6.5, where we have tried to specify boundary conditions using the same procedure as in the linear pressure-drop example (Figure 6.3).

If the reservoir neither had faults with displacement nor holes inside its perimeter, we could use the subrange feature of `fluxside` and `pside` to restrict the boundary conditions to a subset of the global side, i.e., for our particular choice of grid parameters, set

```
bc = fluxside([], G, 'EAST', 5e3*meter^3/day, 4:15, 1:5);
bc = pside  (bc, G, 'WEST', 50*barsa,       7:17,  []);
```

Unfortunately, this will not work properly in the current case, as shown in the middle plot of Figure 6.5. The problem is that `fluxside` and `pside` define their 'east' sides to consist of all faces that only belong to one cell and are categorized to be on the east side of this cell.

To find the faces that are on the perimeter, we need to use expert knowledge. In our case, this amounts to utilizing the fact that the perimeter is defined as those faces that lie on the bounding box of the model. For the Neumann condition we therefore get

```
x = G.faces.centroids(f,1);
[xm,xM] = deal(min(x), max(x));
ff = f(x>xM-1e-5);
bc = addBC(bc, ff, 'flux', (5e3*meter^3/day) ...
             * G.faces.areas(ff)/ sum(G.faces.areas(ff)));
```

Notice how the total flux has been distributed to individual faces according to the face area. The Dirichlet condition can be specified in a similar manner.

Computer exercises:

1. Consider a 2D box with a sink at the midpoint and inflow across the perimeter specified either in terms of a constant pressure or a constant flux. Are there differences in the two solutions, and if so, can you explain why? Hint: use time-of-flight, total travel time, and/or streamlines to investigate the flow pattern.
2. Apply the production setup from Figure 6.4, with hydrostatic boundary conditions and fluids extracted from two cells at the midpoint of the model, to the model depicted in Figure 6.5.

6.4.3 Structured versus unstructured stencils

We have so far only discussed grids that have an underlying structured cell numbering. The two-point schemes can also be applied to fully unstructured and polyhedral grids. To demonstrate this, we use the triangular grid generated from the `seamount` data set that is supplied with MATLAB, see Figure 3.8, scaled to cover a 1×1 km^2 area. Based on this grid, we define a non-rectangular reservoir. The reservoir is assumed to be homogeneous with an isotropic permeability of 100 mD and the resident fluid has the same properties as in the previous examples. A constant pressure of 50 bar is set at the outer perimeter and fluid is drained from a well located at $(450, 500)$ at a constant rate of one pore volume over fifty years. (All details are found in the script `stencilComparison.m`).

We start by generating the triangular grid, which will subsequently be used to define the extent of the reservoir:

```
load seamount
T = triangleGrid([x(:) y(:)], delaunay(x,y));
[Tmin,Tmax] = deal(min(T.nodes.coords), max(T.nodes.coords));
T.nodes.coords = bsxfun(@times, ...
    bsxfun(@minus, T.nodes.coords, Tmin), 1000./(Tmax - Tmin));
T = computeGeometry(T);
```

Next, we generate two Cartesian grids that cover the same domain, one with approximately the same number of cells as the triangular grid and a 10×10 refinement of this grid that will give us a reference solution,

```
G = computeGeometry(cartGrid([25 25], [1000 1000]));
inside = isPointInsideGrid(T, G.cells.centroids);
G = removeCells(G, ~inside);
```

The function `isPointInsideGrid` implements a simple algorithm for finding whether one or more points lie inside the circumference of a grid. First, all boundary faces are extracted and then the corresponding nodes are sorted so that they form a closed polygon. Then, MATLAB's built-in function `inpolygon` can be used to check whether the points are inside this polygon or no.

To construct a radial grid centered around the point at which we will extract fluids, we start by using the same code as on page 68 to generate a set of points inside $[-1, 1] \times [-1, 1]$ that are graded radially towards the origin (see e.g., Figure 3.22),

```
P = [];
for r = exp([-3.5:.2:0, 0, .1]),
   [x,y] = cylinder(r,25); P = [P [x(1,:); y(1,:)]];
end
P = unique([P'; 0 0], 'rows');
```

The points are scaled and translated so that their origin is moved to the point (450,500), from which fluid will be extracted:

```
[Pmin,Pmax] = deal(min(P), max(P));
P = bsxfun(@minus, bsxfun(@times, ...
          bsxfun(@minus, P, Pmin), 1200./(Pmax-Pmin)), [150 100]);
```

Then, we remove all points outside of the triangular grid, before the point set is passed to two grid-factory routines to first generate a triangular and then a Voronoi grid:

```
inside = isPointInsideGrid(T, P);
V = computeGeometry( pebi( triangleGrid(P(inside,:)) ));
```

Once the grids have been constructed, the setup of the remaining part of the model will be the same in all cases. To avoid unnecessary replication of

code, we collect the grids in a cell array and use a simple `for`-loop to set up and simulate each model realization:

```
g = {G, T, V, Gr};
for i=1:4
   rock.poro = repmat(0.2, g{i}.cells.num, 1);
   rock.perm = repmat(100*milli*darcy, g{i}.cells.num, 1);
   hT = simpleComputeTrans(g{i}, rock);
   pv = sum(poreVolume(g{i}, rock));

   tmp = (g{i}.cells.centroids - repmat([450, 500],g{i}.cells.num,[])).^2;
   [~, ind] = min(sum(tmp,2));
   src{i} = addSource(src{i}, ind, -.02*pv/year);

   f = boundaryFaces(g{i});
   bc{i} = addBC([], f, 'pressure', 50*barsa);

   state{i} = incompTPFA(initResSol(g{i},0,1), ...
       g{i}, hT, fluid, 'src', src{i}, 'bc', bc{i}, 'MatrixOutput', true);

   [tof{i},A{i}] = computeTimeOfFlight(state{i}, g{i}, rock,...
       'src', src{i},'bc',bc{i}, 'reverse', true);
end
```

The pressure solutions computed on the four different grids are shown in Figure 6.6 compares, while Figure 6.7 compares the sparsity patterns of the corresponding linear systems for the three coarse grids.

As expected, the Cartesian grid gives a banded matrix consisting of five diagonals that correspond to the each cell and its four neighbors in the cardinal directions. Even though this discretization is not able to predict the complete draw-down at the center (the reference solution predicts a pressure slightly below 40 bar), it captures the shape of the draw-down region quite accurately; the region appears ellipsoidal because of the non-unit aspect ratio in the plot. In particular, we see that the points in the radial plot follow those of the fine-scale reference closely. The spread in the points as $r \to 300$ is not a grid-orientation effect, but the result of variations in the radial distance to the fixed pressure at the outer boundary on all four grids.

The unstructured triangular grid is more refined near the well and is hence able to predict the pressure draw-down in the near-well region more accurately. However, the overall structure of this grid is quite irregular, as can be seen from the sparsity pattern of the linear system shown in Figure 6.7, and the irregularity gives significant grid-orientation effects. This can be seen from the irregular shape of the color contours in the upper part of Figure 6.6 as well as from the spread in the scatter plot. In summary, this grid is not well suited for resolving the radial symmetry of the pressure draw-down in the near-well region. But to be fair, the grid was not generated for this purpose either.

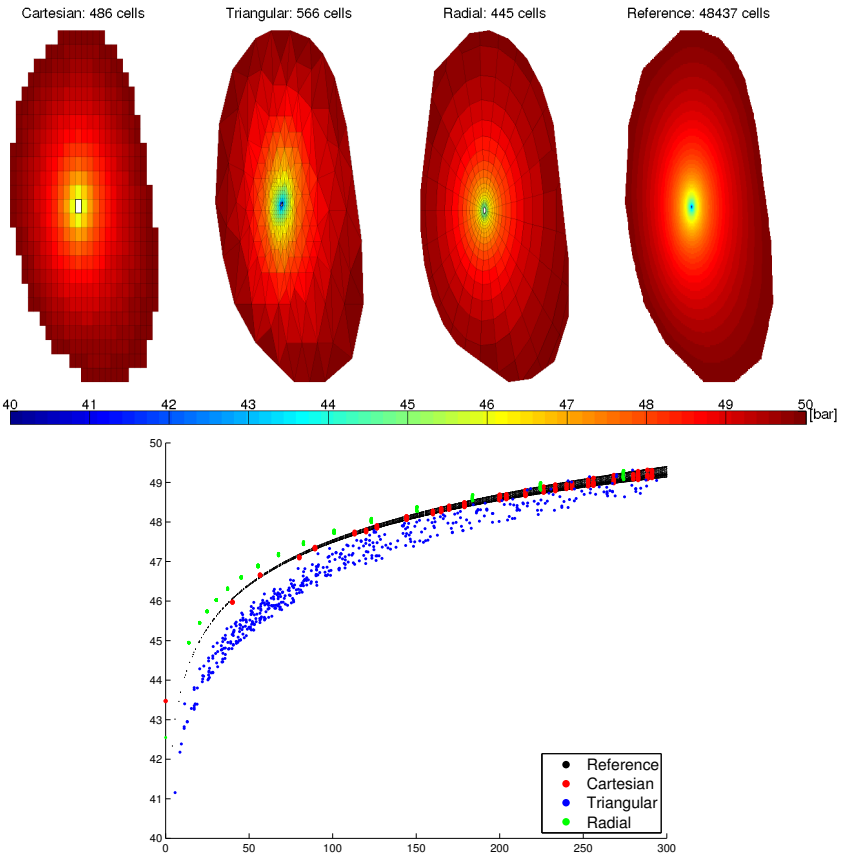


Fig. 6.6. Comparison of the pressure solution for three different grid types: uniform Cartesian, triangular, and a graded radial grid. The scattered points used to generate the triangular domain and limit the reservoir are sampled from the `seamount` data set and scaled to cover a 1×1 km^2 area. Fluids are drained from the center of the domain, assuming a constant pressure of 50 bar at the perimeter.

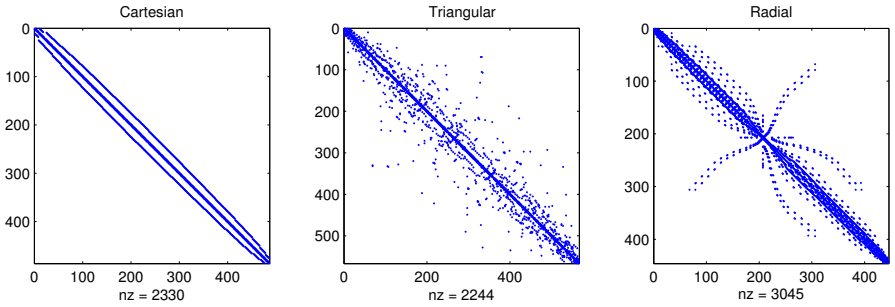


Fig. 6.7. Sparsity patterns for the TPFA stencils on the three different grid types shown in Figure 6.6.

Except for close to the well and close to the exterior boundary, the topology of the radial grid is structured in the sense that each cell has four neighbors, two in the radial direction and two in the angular direction, and the cells are regular trapezoids. This should, in principle, give a banded sparsity pattern provided that the cells are ordered starting at the natural center point and moving outward, one ring at the time. To verify this claim, you can execute the following code:

```
[~,q]=sort(state{3}.pressure);
spy(state{3}.A(q,q));
```

However, as a result of how the grid was generated, by first triangulating and then forming the dual, the cells are numbered from west to east, which explains why the sparsity pattern is so far from being a simple banded structure. While this may potentially affect the efficiency of a linear solver, it has no impact on the accuracy of the numerical approximation, which is good because of the grading towards the well and the symmetry inherent in the grid. Slight differences in the radial profile compared with the Cartesian grid(s) can mainly be attributed to the fact that the source term and the fixed pressure conditions are not located at the exact same positions in the simulations.

In Figure 6.8, we also show the sparsity pattern of the linear system used to compute the reverse time-of-flight from the well and back into the reservoir. Using the default cell ordering, the sparsity pattern of each upwind matrix will appear as a less dense version of the pattern for the corresponding TPFA matrix. However, whereas the TPFA matrices represent an elliptic equation in which information propagates in both directions across cell interfaces, the upwind matrices are based on one-way connections arising from fluxes between pairs of cells that are connected in the TPFA discretization. To reveal the true nature of the system, we can permute the system by either sorting the cell pressures in ascending order (potential ordering) or using the function `dmperm` to compute a Dulmage–Mendelsohn decomposition. As pointed out in Section 6.3, the result is a lower triangular matrix, from which it is simple to see that the unidirectional propagation of information one would expect for a hyperbolic equations having only positive characteristics.

Computer exercises:

1. Compare the sparsity patterns resulting from the potential ordering and use of `dmperm` for both the upwind and the TPFA matrices.
2. Investigate the flow patterns in more details using forward time-of-flight, travel time, and/or streamlines.
3. Replace the boundary conditions by a constant influx, or set pressure values sampled from a radially symmetric pressure solution in an infinite domain.

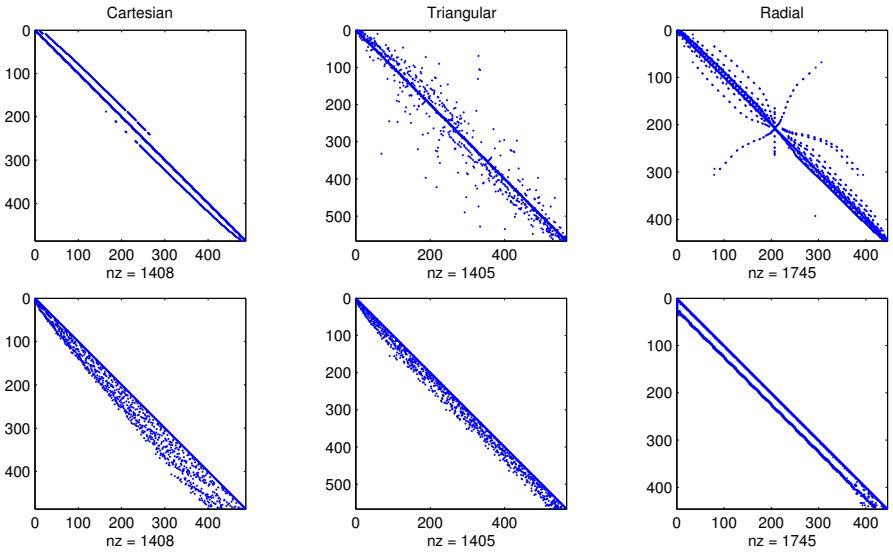


Fig. 6.8. Sparsity patterns for the upwind stencils used to compute time-of-flight on the three different grid types shown in Figure 6.6. In the lower row, the matrices have been permuted to lower-triangular form by sorting the cell pressures in ascending order.

6.4.4 Using Peaceman well models

Whereas it may be sufficient to consider flow driven by sources, sinks, and boundary conditions in many subsurface applications, the key aspect in reservoir simulation is in most cases to predict the amount of fluids that are produced and/or injected from one or more wells. As we saw in Section 5.3.2, flow in and out of a wellbore takes place on a scale that is much smaller than those of a single grid cell in typical sector and field models and is therefore commonly modeled using a semi-analytical model of the form (5.32). In this section, we will go through two examples to demonstrate how such models can be included in the simulation setup using data objects and utility functions introduced in Section 6.1.5. The first example is a highly idealized box model. In the second example we consider a realistic model of a shallow-marine reservoir taken from the SAIGUP study, see Section 2.4.5.

Box reservoir

We consider a reservoir consisting of a homogeneous $500\times500\times25$ m^3 sand box with a isotropic permeability of 100 mD, represented on a regular $20\times20\times5$ Cartesian grid. The fluid is the same as in the examples above. All code lines necessary to set up the model, solve the flow equations, and visualize the results are found in the script `firstWellExample.m`.

Setting up the model is quickly done, once you have gotten familiar with MRST:

```
[nx,ny,nz] = deal(20,20,5);
G = computeGeometry( cartGrid([nx,ny,nz], [500 500 25]) );
rock.perm = repmat(100 .* milli*darcy, [G.cells.num, 1]);
fluid = initSingleFluid('mu', 1*centi*poise,'rho', 1014*kilogram/meter^3);
hT    = computeTrans(G, rock);
```

The reservoir will be produced by a well pattern consisting of a vertical injector and a horizontal producer. The injector is located in the south-west corner of the model and operates at a constant rate of 3000 m^3 per day. The producer is completed in all cells along the upper east rim and operates at a constant bottom-hole pressure of 1 bar (i.e., 10^5 Pascal in SI units):

```
W = verticalWell([], G, rock, 1, 1, 1:nz, 'Type', 'rate', 'Comp_i', 1, ...
                 'Val', 3e3/day(), 'Radius', .12*meter, 'name', 'I');
W = addWell(W, G, rock, nx : ny : nx*ny, 'Type', 'bhp', 'Comp_i', 1, ...
            'Val', 1.0e5, 'Radius', .12*meter, 'Dir', 'y', 'name', 'P');
```

In addition to specifying the type of control on the well ('bhp' or 'rate'), we also need to specify the radius and the fluid composition, which is '1' here since we have a single fluid. After initialization, the array `W` contains two data objects, one for each well:

```
Well #1:                          |       Well #2:
       cells: [5x1 double]        |              cells: [20x1 double]
        type: 'rate'              |               type: 'bhp'
         val: 0.0347              |                val: 100000
           r: 0.1000              |                  r: 0.1000
         dir: [5x1 char]          |                dir: [20x1 char]
          WI: [5x1 double]        |                 WI: [20x1 double]
          dZ: [5x1 double]        |                 dZ: [20x1 double]
        name: 'I'                 |               name: 'P'
       compi: 1                   |              compi: 1
    refDepth: 0                   |           refDepth: 0
        sign: 1                   |               sign: []
```

This concludes the specification of the model. We can now assemble and solve the system

```
gravity reset on;
resSol = initState(G, W, 0);
state  = incompTPFA(state, G, hT, fluid, 'wells', W);
```

The result is shown in Figure 6.9. As expected, the inflow rate decays with the distance to the injector. The flux intensity depicted in the lower-right plot is computed using the following command, which first maps the vector of face fluxes to a vector with one flux per half face and then sums the absolute value of these fluxes to get a flux intensity per cell:

```
cf = accumarray(getCellNoFaces(G), ...
   abs(faceFlux2cellFlux(G, state.flux)));
```

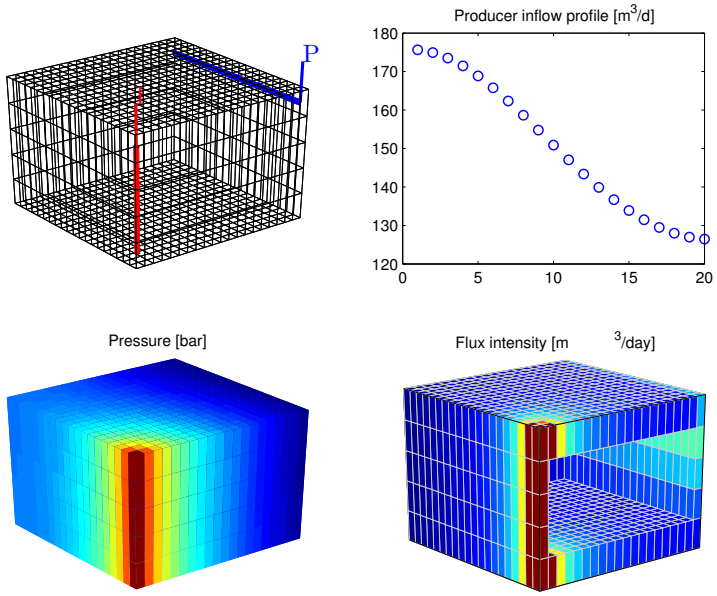


Fig. 6.9. Solution of a single-phase, incompressible flow problem inside a box reservoir with a vertical injector and a horizontal producer.

Shallow-marine reservoir

In the final example, we will return to the SAIGUP model discussed in Section 3.4. This model does not represent a real reservoir, but is one out of a large number of models that were built to be plausible realizations that contain the types of structural and stratigraphic features one could encounter in models of real clastic reservoirs. Continuing from Section 3.4, we simply assume that the grid and the petrophysical model has been loaded and processed. All details are given in the script `saigupWithWells.m`. (The script also explains how to speed up the grid processing by using two C-accelerated routines for constructing a grid from Eclipse input and computing areas, centroids, normals, volumes, etc).

The permeability input is an anisotropic tensor with zero vertical permeability in a number of cells. As a result, some parts of the reservoir may be completely sealed off from the wells. This will cause problems for the time-of-flight solver, which requires that all cells in the model must be flooded after some finite time that can be arbitrarily large. To avoid this potential problem, we assign a small constant times the minimum positive vertical permeability to the grid blocks that have zero cross-layer permeability.

```
is_pos = rock.perm(:, 3) > 0;
rock.perm(~is_pos, 3) = 1e-6*min(rock.perm(is_pos, 3));
```

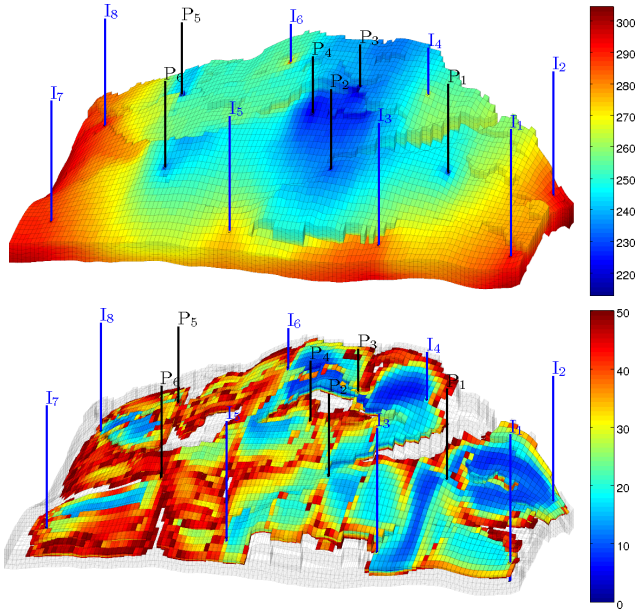


Fig. 6.10. Incompressible, single-phase simulation of the SAIGUP model. The upper plot shows pressure distribution, and the lower plot shows cells with total travel time less than fifty years.

Similar safeguards are implemented in most commercial simulators.

The reservoir is produced from six producers spread throughout the middle of the reservoir; each producer operates at a fixed bottom-hole pressure of 200 bar. Pressure support is provided by eight injectors located around the perimeter, each operating at a prescribed and fixed rate. As in the previous example, the wells are described using a Peaceman model. For simplicity, all wells chosen to be vertical and are assigned using the logical ij sub-index available in the corner-point format. The following code specifies the injectors:

```
nz = G.cartDims(3);
I = [ 3,  20,   3,  25,   3,  30,   5,  29];
J = [ 4,   3,  35,  35,  70,  70,113,113];
R = [ 1,   3,   3,   3,   2,   4,   2,   3]*500*meter^3/day;
W = [];
for i = 1 : numel(I),
  W = verticalWell(W, G, rock, I(i), J(i), 1:nz, 'Type', 'rate',  ...
                   'Val', R(i), 'Radius', .1*meter, 'Comp_i', 1,  ...
                   'name', ['I$_{', int2str(i), '}$']);
end
```

The producers are specified in the same way. Figure 6.10 shows the well positions and the pressure distribution. We see a clear pressure buildup along the

east, south, and west rim of the model. Similarly, there is a pressure drawdown in the middle of the model around producers P2, P3, and P4. The total injection rate is set so that one pore volume will be injected in a little less than forty years.

Although this is a single-phase simulation, let us for a while think of our setup in terms of injection and production of different fluids (since the fluids have identical properties, we can think of a 'blue' fluid being injected into a 'black' fluid). In an ideal situation, one would wish that the 'blue' fluid would sweep the whole reservoir before it breaks through to the production wells, as this would maximize the displacement of the 'black' fluid. Even in the simple quarter five-spot examples in Section 6.4.1 (see Figure 6.2), we saw that this was not the case, and one cannot expect that this will happen here, either. The lower plot in Figure 6.10 shows all cells in which the total travel time (sum of forward and backward time-of-flight) is less than fifty years. By looking at such a plot, on can get a quite a good idea of regions in which there is very limited communication between the injectors and producers (i.e., areas without colors). If this was a multiphase flow problem, these areas would typically contain bypassed oil and be candidates for infill drilling or other mechanisms that would improve the volumetric sweep.

Computer exercises:

1. Change the parameter 'Dir' from 'y' to 'z' in the box example and rerun the case. Can you explain why you get a different result?
2. Switch the injector in the box example to be controlled by a bottom-hole pressure of 200 bar. Where would you place the injector to maximize production rate if you can only complete it in five cells?
3. Consider the SAIGUP model: can you improve the well placement and/or the distribution of fluid rates. Hint: is it possible to utilize time-of-flight information?

7

Single-Phase Solvers Based on Automatic Differentiation

In the previous chapter, we outlined and explained details of the functionality MRST offers for simulation of *incompressible, single-phase problems*. Later in the book we will come back to how these methods and tools can be extended to time-dependent multiphase problems based on various operator-splitting formulations in which flow and transport are computed in different substeps. Operator splitting is one of the main approaches used for simulating flow in porous media. The other approach, which is the predominantly used in industry, is based on so-called fully-implicit discretizations in which flow and transport are solved as a coupled system. This approach is very robust and particularly useful for compressible problems with small time constants or strong coupling between different types of flow mechanisms.

In this chapter, we will discuss fully-implicit discretizations of compressible, single-phase problems. In doing so, we will also introduce you to **automatic differentiation (AD)**, which will enable you to numerically evaluate the derivative of a function specified by a computer program directly without first having to derive a set of underlying analytical expressions. The combination of this concept with the highly vectorized and interactive scripting language of Matlab and MRST's flexible grid structure is in our opinion the main reason why MRST has proved to be a particularly efficient tool for rapid prototyping of new models and computational methods. To substantiate this claim, we will in the following demonstrate how the discrete differential operators introduced in Section 5.4.2, and similar discrete averaging operators, can be used as powerful abstractions that will enable you to keep the physics at the forefront in your code and rapidly explore alternative methods for discretizing and linearizing nonlinear flow equations.

7.1 Implicit discretization

As our basic model, we will in the following mainly be concerned with the single-phase continuity equation

$$\frac{\partial}{\partial t}(\phi\rho) + \nabla \cdot (\rho\vec{v}) = q, \qquad \vec{v} = -\frac{\mathsf{K}}{\mu}\left(\nabla p - g\rho\nabla z\right). \tag{7.1}$$

The primary unknown is usually the fluid pressure p and additional equations are supplied to provide relations between p and the other quantities in the equation, e.g., by specifying $\phi = \phi(p)$, an equation-of-state $\rho = \rho(p)$ for the fluid, and so on; see the discussion in Section 5.2.

Using the discrete operators introduced in Section 5.4.2, the basic implicit discretization of (7.1) reads

$$\frac{(\boldsymbol{\phi\rho})^{n+1} - (\boldsymbol{\phi\rho})^{n}}{\Delta t^n} + \mathtt{div}(\boldsymbol{\rho v})^{n+1} = \boldsymbol{q}^{n+1}, \tag{7.2a}$$

$$\boldsymbol{v}^{n+1} = -\frac{\boldsymbol{K}}{\boldsymbol{\mu}^{n+1}}\left[\mathtt{grad}(\boldsymbol{p}^{n+1}) - g\boldsymbol{\rho}^{n+1}\mathtt{grad}(\boldsymbol{z})\right]. \tag{7.2b}$$

Here, $\boldsymbol{\phi} \in \mathbb{R}^{n_c}$ denotes the vector porosity values per cell, $\boldsymbol{v}$ the vector of fluxes per face, and so on, and the superscript refers to discrete times at which one wishes to compute the unknown reservoir states and Δt denotes the time between two consecutive points in time.

In many cases of practical interest it is possible to simplify (7.2). For instance, if the fluid is only slightly compressible, several terms can be neglected so that the nonlinear equation reduces to an equation that is linear in the unknown p^{n+1},

$$\frac{\boldsymbol{p}^{n+1} - \boldsymbol{p}^{n}}{\Delta t^n} - \frac{1}{c_t\boldsymbol{\phi}\mu}\mathtt{div}\big(\boldsymbol{K}\,\mathtt{grad}(\boldsymbol{p}^{n+1})\big) = \boldsymbol{q}^{n}. \tag{7.3}$$

However, this is not always possible and for generality, we will in the following assume that ϕ and ρ depend nonlinearly on p so that (7.2) gives rise to a (highly) nonlinear system of equations that needs to be solved in each time step. For non-Newtonian fluids, the viscosity will also depend on the velocity, which will add further nonlinearity to the system and generally make it harder to solve. A simple example is a power law of the form $\mu(|\vec{v}|) = \mu_0|\vec{v}|^{\alpha-1}$, where α is a parameter that is determined experimentally.

In the following, we will mainly work with discretized equations written in residual form. As an example, the residual form of (7.3) reads

$$\boldsymbol{p}^{n+1} - \frac{\Delta t^n}{c_t\boldsymbol{\phi}\mu}\mathtt{div}\big(\boldsymbol{K}\,\mathtt{grad}(\boldsymbol{p}^{n+1})\big) - \boldsymbol{p}^{n} - \Delta t^n\boldsymbol{q}^{n} = 0.$$

After spatial discretization, we can write the resulting system of nonlinear equations in the short vector-form as

$$\boldsymbol{F}(\boldsymbol{x}^{n+1}; \boldsymbol{x}^{n}) = \boldsymbol{0}, \tag{7.4}$$

where $\boldsymbol{x}^{n+1}$ is the vector of unknown state variables at the next time step and the vector of current states $\boldsymbol{x}^{n}$ can be seen as a parameter.

7.2 Automatic differentiation

Nonlinear systems of discrete equations arising from the discretization of (partial) differential equations are typically solved by Newton's method. In most cases, this means that the main computational cost of solving a nonlinear PDE lies in solving the linear systems involved in each Newton iteration. For a discretized equation on vector form, $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$, the $(i+1)$-th iteration approximation $\boldsymbol{x}^{i+1}$ is obtained from

$$\frac{\partial \boldsymbol{F}(\boldsymbol{x}^i)}{\partial \boldsymbol{x}^i}\delta\boldsymbol{x}^{i+1} = -\boldsymbol{F}(\boldsymbol{x}^i), \quad \boldsymbol{x}^{i+1} \leftarrow \boldsymbol{x}^i + \delta\boldsymbol{x}^{i+1}. \tag{7.5}$$

Here, $\boldsymbol{J}(\boldsymbol{x}^i) = \partial\boldsymbol{F}(\boldsymbol{x}^i)/\partial\boldsymbol{x}^i$ is the Jacobian matrix, while we refer to $\delta\boldsymbol{x}^{i+1}$ as the *Newton update* at iteration step number $i+1$. The Newton process will under certain smoothness and differentiability requirements exhibit quadratic convergence. This is, however, crucially dependent on a sufficiently accurate Jacobian matrix. Typically, obtaining the Jacobian matrix can be broken down to differentiation of elementary operations and functions. Nevertheless, if $\boldsymbol{F}$ represents a set of complex equations, analytical derivation and subsequent coding of the Jacobian can be very time-consuming and prone to errors and debugging. On the other hand, computing Jacobians is a good candidate for automation since it follows a set of fixed rules and should therefore in principle be *straightforward in principle*.

Automatic differentiation is a technique that exploits the fact that any computer code, regardless of complexity, can be broken down to a limited set of arithmetic operations ($+$, $-$, $*$, $/$, etc), and, in our case, more or less elementary MATLAB functions (`exp`, `sin`, `power`, `interp`, etc). In automatic differentiation (AD) the key idea is to keep track of quantities and their derivatives simultaneously; every time an operation is applied to a quantity, the corresponding differential operation is applied to its derivative. Consider a scalar primary variable x and a function $f = f(x)$. Their AD-representations would then be the pairs $\langle x, 1\rangle$ and $\langle f, f_x\rangle$, where f_x is the derivative of f w.r.t. x (and 1 is the derivative of x w.r.t. x). Accordingly, the action of the elementary operations and functions must be defined for such pairs, e.g.,

$$\begin{aligned}
\langle f, f_x\rangle + \langle g, g_x\rangle &= \langle f+g, f_x+g_x\rangle\,,\\
\langle f, f_x\rangle * \langle g, g_x\rangle &= \langle fg, fg_x + f_x g\rangle\,,\\
\langle f, f_x\rangle \,/\, \langle g, g_x\rangle &= \left\langle \frac{f}{g}, \frac{f_x g - f g_x}{g^2}\right\rangle\\
\exp(\langle f, f_x\rangle) &= \langle \exp(f), \exp(f) f_x\rangle\,,\\
\sin(\langle f, f_x\rangle) &= \langle \sin(f), \cos(f) f_x\rangle\,.
\end{aligned}$$

In addition to this, one needs to use the chain rule to accumulate derivatives; that is, if $f(x) = g(h(x))$, then $f_x(x) = \frac{dg}{dh}h_x(x)$. This more or less summarizes the key idea behind automatic differentiation, the remaining (and difficult

part) is how to implement the idea as efficient computer code that has a low user-threshold and minimal computational overhead.

As the above example illustrates, it is straightforward to write down all elementary rules needed to differentiate a program or piece of code. To be useful, however, it is important that these rules are not implemented as standard functions, in which case you would need to write something like `myPlus(a, myTimes(b,c))` when you want to evaluate $a + bc$. An elegant solution to this is the use of classes and operator overloading. When MATLAB encounters an expression `a+b`, the meaning of "+" is dependent on the nature of `a` and `b`. In other words, there may be multiple functions `plus` defined, and the one to choose is determined from which classes `a` and `b` belongs to. A nice introduction to how this is can be implemented in MATLAB is given by Neidinger [46].

7.3 Automatic differentiation in MRST

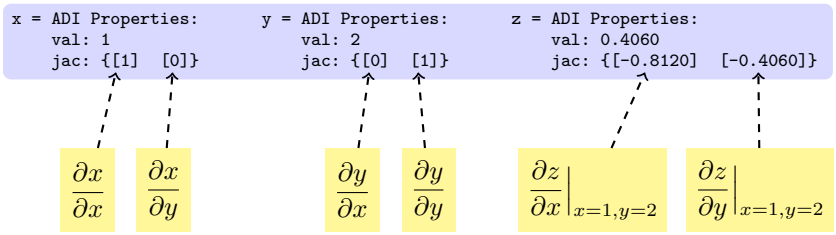
There are many automatic differentiation libraries for MATLAB, e.g., ADiMat [57, 12], ADMAT [17, 59], MAD [58, 56, 31], or from MATLAB Central [30, 43]. The AD class in MRST uses operator overloading as suggested in [46] and uses a relatively simple forward accumulation, but differs from other libraries in a subtle, but important way. Instead of working with a single Jacobian of the full discrete system as one matrix, MRST uses a list of matrices that represent the derivatives with respect to different variables, which will constitute sub-blocks in the Jacobian of the full system. The reason for this choice is two-fold: computational performance and user utility. In typical simulation, and particularly for complex model, the mathematical model will consist of several equations (continuum equations, Darcy's law, equations of state, other constitutive relationships, control equations for wells, etc) that have different characteristics and play different roles in the overall equation system. Although we are using fully implicit discretizations in which one seeks to solve for all state variables simultaneously, we may still want to manipulate parts of the full equation system that e.g., represents specific sub-equations. This is not practical if the Jacobian of the system is represented as a single matrix; manipulating subsets of large sparse matrices is of currently not very efficient in MATLAB, and keeping track of the necessary index sets may also be quite cumbersome from a user's point-of-view. Accordingly, our current choice is to let the MRST AD-class represent the derivatives of different primary variable (e.g., pressure, saturations, bottom-hole-pressures, ...) as a list of matrices.

In the rest of the section, we will instead go through two simple examples that demonstrate how the AD class works, before we move on to demonstrate how automatic differentiation can be used to set up simulations in a (surprisingly) few number of code lines.

Example 7.1. As a first example, let us say we want to compute the expression $z = 3e^{-xy}$ and its partial derivatives $\partial z/\partial x$ and $\partial z/\partial y$ for the values $x = 1$ and $y = 2$. This is done with the following two lines:

```
[x,y] = initVariablesADI(1,2);
z = 3*exp(-x*y)
```

The first line tells MRST that `x` and `y` are independent variables and initialize their values. The second line is what you normally would write in MATLAB to evaluate the given expression. After the second line has been executed, you have three AD variables (pairs of values and derivatives):



If we now go on computing with these variables, each new computation will lead to a result that contains the value of the computation as well as the derivatives with respect to x and y.

Let us look a bit in detail on what is happening behind the curtain. We start by observing that the operation `3*exp(-x*y)` in reality consists of a sequence of elementary operations: $-$, $*$, exp, and $*$, executed in that order. In MATLAB, this corresponds to the following sequence of call to elementary functions

```
u = uminus(x);
v = mtimes(u,y);
w = exp(u);
z = mtimes(3,w);
```

To see this, you can enter the command into a file, set a breakpoint in front of the assignment to `z`, and use the 'Step in' button to step through all details. The AD class overloads these three functions by new functions that have the same names, but operate on an AD pair for `uminus` and `exp`, and on two AD pairs or a combination of a double and an AD pair for `mtimes`. Figure 7.1 gives an overview of the sequence of calls that is invoked within the AD implementation to evaluate `3*exp(-x*y)` when `x` and `y` are AD variables. The observant reader may have noticed that some computational saving could be obtained here if we had been careful to replace the call to matrix multiply (`*=mtimes`) by a call to vector multiply (`.*=times`), which are mathematically equivalent for scalar quantities.

As you can see from the above example, use of automatic differentiation will give rise to a whole new set of function calls that are not executed if one

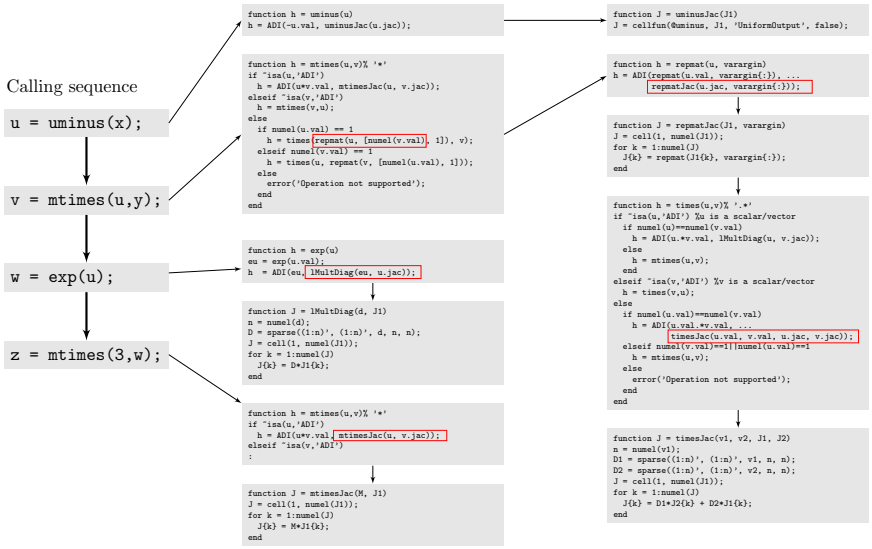


Fig. 7.1. Complete set of functions invoked to evaluate $3*\texttt{exp}(-\texttt{x}*\texttt{y})$ when `x` and `y` are AD variables. For brevity, we have not included details of the constructor function `ADI(val,Jac)`, which constructs an AD pair with the value `val` and list of Jacobi matrices `Jac`.

only wants to evaluate a mathematical expression and not find its derivatives. Apart from the cost of the extra code lines one has to execute, user-defined classes are fairly new in MATLAB and there is still some overhead in using class objects and accessing their properties (e.g., `val` and `jac`) compared to the built-in `struct`-class. The reason why AD still pays off in most examples, is that the cost of generating derivatives is typically much smaller than the cost of the solution algorithms they will be used in, in particular when working with equations systems consisting of large sparse matrices with more than one row per cell in the computational grid. However, one should still seek to limit the number of calls involving AD-class functions (including the constructor). We let the following example be a reminder that vectorization is of particular importance when using AD classes in MRST:

Example 7.2. To investigate the efficiency of vectorization versus serial execution of the AD objects in MRST, we consider the inner product of two vectors

```
z = x.*y;
```

that have been initialized with random numbers. We will compare the cost of using the overloaded version of vector multiply (`.*`) to that of a standard `for`-loop with either matrix multiply (`mtimes`) or vector multiply (`times`) for the scalar multiplications. For comparison, we also include the cost of computing

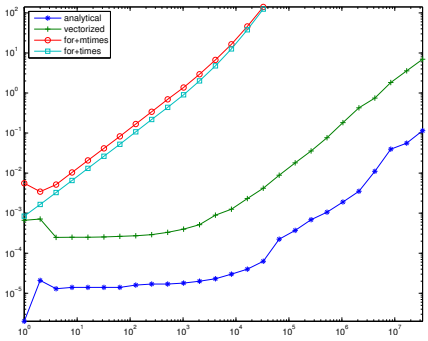


Fig. 7.2. Comparison of cost of computing `z=x*v` and derivatives as function of the number of elements in the vectors.

z and its derivatives $z_x = y$ and $z_y = x$ directly using standard MATLAB vectors of doubles:

```
[n,t1,t2,t3,t4] = deal(zeros(m,1));
for i = 1:m
    n(i) = 2^(i-1);
    xv = rand(n(i),1); yv=rand(n(i),1);
    [x,y] = initVariablesADI(xv,yv);
    tic, z = xv.*yv; zx=yv; zy = xv;          t1(i)=toc;
    tic, z = x.*y;                            t2(i)=toc;
    tic, for k =1:n(i), z(k)=x(k)*y(k); end;  t3(i)=toc;
    tic, for k =1:n(i), z(k)=x(k).*y(k); end; t4(i)=toc;
end
```

Figure 7.2 shows a plot of the corresponding runtimes as function of the number elements in the vector. For this simple function, using AD is a factor 20-40 times more expensive than using direct evaluation of z and the analytical expressions for z_x and z_y. Using vector multiply instead of matrix multiply inside the `for`-loop reduces the cost by 30% on average, but the factor diminishes as the number of elements increases in the vector. In either case, using a loop will on average be more than three orders more expensive than using vectorization.

While `for`-loops in many cases will be quite effectively in MATLAB (contrary to what is common belief), one should of course always try to avoid loops that call functions with non-negligible overhead. The AD class in MRST has been designed to work on long vectors and lists of (sparse) Jacobian matrices and has not been optimized for scalar variables. As a result, there is considerable overhead when working with small AD objects.

Beyond the examples and the discussion above, we will not go more into details about the technical considerations that lie behind the implementation of AD in MRST. If you want a deeper understanding of how the AD class

works, the source code is fully open, so you are free to dissect the details to the level of your own choice.

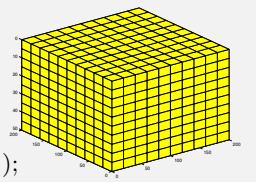
7.4 An implicit single-phase solver

In this section we present step-by-step how one can use the AD class outlined above to implement an implicit solver for the compressible, single-phase (7.1). In particular, we will introduce discrete averaging and spatial differentiation operators that will enable us to write the discretized equations in an abstract residual form that resembles the semi-continuous form of the implicit discretization in (7.2) and is devoid of details about the grid. Starting from this residual form, it is relatively simple to obtain a linearization using automatic differentiation and set up a Newton iteration.

7.4.1 Model setup and initial state

For simplicity, we consider a homogeneous box-model:

```
[nx,ny,nz] = deal( 10,  10,  10);
[Lx,Ly,Lz] = deal(200, 200, 50);
G = cartGrid([nx, ny, nz], [Lx, Ly, Lz]);
G = computeGeometry(G);

rock.perm = repmat(30*milli*darcy, [G.cells.num, 1]);
rock.poro = repmat(0.3 , [G.cells.num, 1]);
```



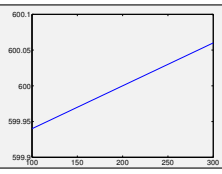
As you will see later, we will hardly see any details about the grid in the following. The main reason for using an idealized model like this, and not a more realistic corner-point model like the SAIGUP or the Norne models discussed in Sections 3.4 and 3.3.1, is that by using a highly idealized reservoir geometry we avoid the complexity of picking representative and reasonable well locations.

We assume a constant rock compressibility c_r. Accordingly, the pore volume `pv` of a grid cell obeys the differential equation $c_r \mathrm{pv} = d\mathrm{pv}/dp$ or

$$\mathrm{pv}(p) = \mathrm{pv}_r e^{c_r(p-p_r)}, \tag{7.6}$$

where pv_r is the pore volume at reference pressure p_r. To define the relation between pore volume and pressure, we use an anonymous function:

```
cr   = 1e-6/barsa;
p_r  = 200*barsa;
pv_r = poreVolume(G, rock);

pv   = @(p) pv_r .* exp( cr * (p - p_r) );
```



```
show = true( G.cells.num ,1);
cellInx = sub2ind( G.cartDims , ...
   [I-1; I-1; I; I;   I(1:2)-1], ...
   [J  ; J;   J; J;   nperf +[2;2]], ...
   [K-1; K;   K; K-1; K(1:2)-[0; 1]]);
show( cellInx ) = false;
plotCellData (G, p_init / barsa , show ,'EdgeColor','k');
plotWell (G,W, 'height',10);
view(-125,20), camproj perspective
```

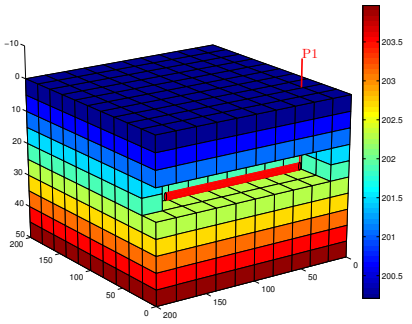


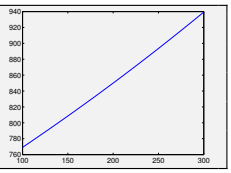
Fig. 7.3. Model with initial pressure and single horizontal well.

The fluid is assumed to have a constant viscosity, $\mu = 5$ cP. As for the rock, we assume a constant fluid compressibility c resulting in the differential equation $c\rho = d\rho/dp$ for the fluid density. Accordingly,

$$\rho(p) = \rho_r e^{c(p-p_r)}, \tag{7.7}$$

where ρ_r =850 kg/m^3 is the density at reference pressure p_r. With this set, we can define the equation of state for the fluid

```
mu    = 5*centi*poise;
c     = 1e-3/barsa;
rho_r = 850*kilogram/meter^3;
rhoS  = 750*kilogram/meter^3;
rho   = @(p) rho_r .* exp( c * (p - p_r) );
```



The assumption of constant compressibility will only hold for a limited range of temperatures. Surface conditions are not inside the validity range of the constant compressibility assumption. We therefore set the fluid density ρ_S at surface conditions separately since we will need it later to evaluate surface volume rate in our model of the well, which here is a horizontal wellbore perforated in eight cells::

```
nperf = 8;
I = repmat(2, [nperf, 1]);
J = (1:nperf).'+1;
K = repmat(5, [nperf, 1]);
cellInx = sub2ind(G.cartDims, I, J, K);
W = addWell([ ], G, rock, cellInx, 'Name', 'producer', 'Dir', 'x');
```

Assuming the reservoir is initially at equilibrium implies that we must have $dp/dz = g\rho(p)$. In this simple setting, this differential equation can be solved analytically, but for demonstration purposes, we will use one of MATLAB's built-in ODE-solvers to compute the hydrostatic distribution numerically, relative to a fixed datum point $p(z_0) = p_r$, where we without lack of generality have set $z_0 = 0$ since the reservoir geometry is defined relative to this height:

```
gravity reset on, g = norm(gravity);
[z_0, z_max] = deal(0, max(G.cells.centroids(:,3)));
equil  = ode23(@(z,p) g .* rho(p), [z_0, z_max], p_r);
p_init = reshape(deval(equil, G.cells.centroids(:,3)), [], 1);
```

This finishes the model setup, and at this stage we plot the reservoir with well and initial pressure (see Figure 7.3)

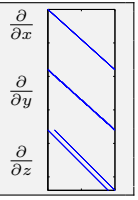
7.4.2 Discrete operators and equations

We are now ready to discretize the model. Assuming no-flow boundary conditions (except at the well), we restrict the equations to the interior faces of the grid. To derive the spatial discretization, we will use the MRST-function `computeTrans`, which we have seen in Section 6.2 computes the half transmissibilities associated with the two-point flux approximation (TPFA). This means that we need to take the harmonic average to obtain the face-transmissibilities, i.e., for neighboring cells i and j, $T_{ij} = (T_{i,j}^{-1} + T_{j,i}^{-1})^{-1}$ as in (5.49).

```
N  = double(G.faces.neighbors);
intInx = all(N ~= 0, 2);
N  = N(intInx, :);                    % Interior neighbors
hT = computeTrans(G, rock);           % Half-transmissibilities
cf = G.cells.faces(:,1);
nf = G.faces.num;
T  = 1 ./ accumarray(cf, 1 ./ hT, [nf, 1]); % Harmonic average
T  = T(intInx);                       % Restricted to interior
```

The final ingredients for the discretization are discrete functions for the divergence and gradient differential operators. As seen in Section 5.4.2, the discrete version of the divergence operator is a linear mapping from the set of faces to the set of cells, and for a flux field, it sums the outward fluxes for each cell. The discrete gradient operator maps from the set of cells to the set of faces, and for a pressure-field, it computes the pressure increase between neighboring cells of each face.In MATLAB notation with N defined as above, it follows that $\texttt{grad}(\boldsymbol{x}) = \boldsymbol{x}(N(:,2)) - \boldsymbol{x}(N(:,1)) = \boldsymbol{C}\boldsymbol{x}$, where the matrix $\boldsymbol{C}$ is defined in the code below. As a linear mapping, the discrete `div`-function is simply the negative transpose of `grad`; this follows from the discrete version of the Gauss-Green theorem, (5.55). Bellow, we will in addition define an *average*-mapping which for each face takes the average value of the neighboring cells

```
n = size(N,1);
C = sparse([(1:n)'; (1:n)'], N, ...
          ones(n,1)*[-1 1], n, G.cells.num);
grad = @(x) C*x;
div  = @(x) -C'*x;
avg  = @(x) 0.5 * (x(N(:,1)) + x(N(:,2)));
```



Having defined the necessary discrete operators, we are in a position to use the basic implicit discretization from (7.2). We start with Darcy's law (7.2b), which for each face f can be written

$$\vec{v}[f] = -\frac{\boldsymbol{T}[f]}{\mu}\left(\mathtt{grad}(\boldsymbol{p}) - g\,\boldsymbol{\rho}_a[f]\,\mathtt{grad}(\boldsymbol{z})\right), \tag{7.8}$$

where the density at the interface is evaluated using arithmetic average:

$$\boldsymbol{\rho}_a[f] = \tfrac{1}{2}\left(\boldsymbol{\rho}[N_1(f)] + \boldsymbol{\rho}[N_2(f)]\right) \tag{7.9}$$

Similarly, we can write the continuity equation for each cell c as

$$\frac{1}{\Delta t}\Big(\left(\boldsymbol{\phi}(\boldsymbol{p})[c]\,\boldsymbol{\rho}(\boldsymbol{p})[c]\right)^{n+1} - \left(\boldsymbol{\phi}(\boldsymbol{p})[c]\,\boldsymbol{\rho}(\boldsymbol{p})[c]\right)^{n}\Big) + \mathtt{div}(\boldsymbol{\rho}_a\boldsymbol{v})[c] = \boldsymbol{0}. \tag{7.10}$$

In MRST, the two residual equations (7.8) and (7.10) are implemented as anonymous functions of pressure:

```
gradz = grad(G.cells.centroids(:,3));
v  = @(p)  -(T/mu).*( grad(p) - g*avg(rho(p)).*gradz );

presEq = @(p,p0,dt)  (1/dt)*(pv(p).*rho(p) - pv(p0).*rho(p0)) ...
                        + div( avg(rho(p)).*v(p) );
```

In the code above, `p0` is the pressure field at the *previous* time step (i.e., $\boldsymbol{p}^{n-1}$), while `p` is the pressure at the *current* time step ($\boldsymbol{p}^{n}$). Having defined the discrete expression for Darcy-fluxes, we can check that this is in agreement with our initial pressure field, i.e., we compute `norm(v(p_init))*day`. The result is $1.5 \times 10^{-6}\,\mathrm{m}^3/\mathrm{day}$, which should convince us that the initial state of the reservoir is sufficiently close to equilibrium.

7.4.3 Well model

To include source-terms in the pressure equation arising from the production well, we need to define an expression for flow rate in each cell that the well is connected to the reservoir (which we will refer to as well connections). Assuming instantaneous flow in the well, we only need to consider hydrostatic pressure drop, which we include for completeness even though the well is horizontal in this particular case. Approximating the fluid density in the well as constant (computed at bottom-hole pressure), the pressure $\boldsymbol{p}_c[w]$ in connection w of well $N_w(w)$ is given by

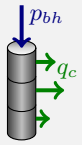
$$\boldsymbol{p}_c[w] = \boldsymbol{p}_{bh}[N_w(w)] + g\,\boldsymbol{\Delta z}[w]\,\rho(\boldsymbol{p}_{bh}[N_w(w)]), \tag{7.11}$$

where $\boldsymbol{\Delta z}[w]$ is the vertical distance from bottom-hole to the connection. To relate the pressure at the well connection to the average pressure inside the grid cell, we will use the standard Peaceman model introduced in Section 5.3.2. Using the well-indices provided in `W`, the mass flow-rate at connection c is given

$$\boldsymbol{q}_c[w] = \frac{\rho(\boldsymbol{p}[N_c(w)])}{\mu}\,\mathrm{WI}[w]\,\big(\boldsymbol{p}_c[w] - \boldsymbol{p}[N_c(w)]\big), \tag{7.12}$$

where $\boldsymbol{p}[N_c(w)]$ is the pressure in the cell $N_c(w)$ surrounding connection w. In MRST, this model reads:

```
wc = W(1).cells; % connection grid cells
WI = W(1).WI;    % well−indices
dz = W(1).dZ;    % depth relative to bottom−hole

p_conn = @(bhp)     bhp + g*dz.*rho(bhp); %connection pressures
q_conn = @(p, bhp)  WI .* (rho(p(wc)) / mu) .* (p_conn(bhp) − p(wc));
```



We will also include the total volumetric well-rate at surface conditions as a free variable. This is simply given by summing all mass well-rates and dividing by the surface density:

```
rateEq = @(p, bhp, qS)  qS−sum(q_conn(p, bhp))/rhoS;
```

With free variables `p`, `bhp`, and `qS`, we are now lacking exactly one equation to close the system. This equation should account for *boundary conditions* in the form of a well-control. Here, we choose to control the well with specified bottom-hole pressure set to 100 bar

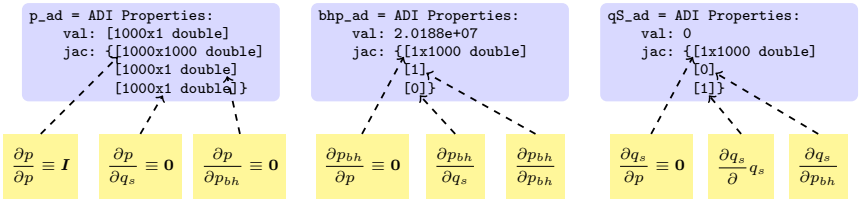
```
ctrlEq = @(bhp) bhp−100*barsa;
```

7.4.4 The simulation loop

We are now in position to start coding our simulation loop. We start by initializing our AD variables (we use `_ad`-subscript to distinguish them from doubles). The initial bottom-hole pressure is set to the corresponding grid cell pressure.

```
[p_ad, bhp_ad, qS_ad] = initVariablesADI(p_init, p_init(wc(1)), 0);
```

This gives the following AD pairs that make up the unknowns in our system:



To solve the global flow problem, we will have to stack all the equations into one big system for which we can compute the Jacobian and perform a Newton update. We therefore set indices for easy access to individual variables in the stack:

```
[p_ad, bhp_ad, qS_ad] = initVariablesADI(p_init, p_init(wc(1)), 0);
nc = G.cells.num;
[pIx, bhpIx, qSIx] = deal(1:nc, nc+1, nc+2);
```

Next, we set parameters that will control the time steps in the simulation and the iterations in the Newton solver:

```
numSteps = 52;                % number of time-steps
totTime = 365*day;            % total simulation time
dt      = totTime / numSteps; % constant time step
tol     = 1e-5;               % Newton tolerance
maxits  = 10;                 % max number of Newton its
```

We will store simulation results from all time steps in a structure `sol`, which we allocate before the main loop for efficiency and initialize so that the first entry is the initial state of the reservoir:

```
sol    = repmat(struct('time', [ ], 'pressure', [ ], 'bhp', [ ], ...
                       'qS', []), [numSteps + 1, 1]);
sol(1) = struct('time', 0, 'pressure', double(p_ad), ...
                'bhp', double(bhp_ad), 'qS', double(qS_ad));
```

We now have all we need to set up the time-stepping algorithm, which will consist of an outer and an inner loop. The outer loop updates the time step, advances the solution forward one step in time, and stores the result in the `sol` structure. This procedure is repeated until we reach the desired final time:

```
t = 0; step = 0;
while t < totTime,
   t = t + dt; step = step + 1;
   fprintf('\nTime step %d: Time %.2f -> %.2f days\n', ...
           step, convertTo(t - dt, day), convertTo(t, day));
   % Newton loop
   resNorm = 1e99;
   p0  = double(p_ad); % Previous step pressure
   nit = 0;
   while (resNorm > tol) && (nit <= maxits)
      : % Newton update
      :
      resNorm = norm(res);
      nit     = nit + 1;
      fprintf('  Iteration  %3d:  Res = %.4e\n', nit, resNorm);
   end
   if nit > maxits, error('Newton solves did not converge')
   else % store solution
      sol(step+1) = struct('time', t, 'pressure', double(p_ad), ...
                           'bhp', double(bhp_ad), 'qS', double(qS_ad));
   end
end
```

The inner loop performs the Newton iteration by computing and assembling the Jacobian of the global system and solving the linearized residual equation to compute an iterative update. The first step to this end is to evaluate the residual for the flow pressure equation and add source terms from wells:

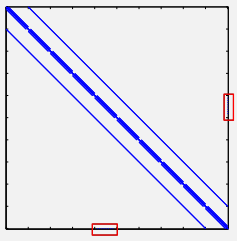
```
eq1     = presEq(p_ad, p0, dt);
eq1(wc) = eq1(wc) - q_conn(p_ad, bhp_ad);
```

Most of the lines we have implemented so far are fairly standard, except perhaps for the definition of the residual equations as anonymous functions, and equivalent statements can be found in almost any computer program solving this type of time-dependent equation by an implicit method. Now, however, comes what is normally the tricky part: linearization of each equation that make up the whole model and assembly of the resulting Jacobian matrices to generate the Jacobian for the full system. And here you have the magic of automatic differentiation implemented in MRST – *you do not have to do this at all!* The computer code necessary to evaluate all the Jacobians has been defined implicitly by the functions in the AD class that overload the elementary operators used to define the residual equations. The calling sequence is obviously more complex than the one depicted in Figure 7.1, but the operators used are in fact only the three elementary operators $+$, $-$, and $*$ applied to scalars, vectors, and matrices, as well as element-wise division by a scalar ($/$). When the residuals are evaluated using the anonymous functions defined above, the AD library will also evaluate the derivatives of each equation with respect to each independent variable and collect the corresponding sub-Jacobians in a list. To form the full system, we simply evaluate the residuals of the remaining equations (the rate equation and the equation for well control) and concatenate the three equations into a cell array:

```
eqs = {eq1, rateEq(p_ad, bhp_ad, qS_ad), ctrlEq(bhp_ad)};
eq  = cat(eqs{:});
```

In doing this, the AD library will correctly combine the various sub-Jacobians and set up the Jacobian for the full system. Then, we can extract this Jacobian, compute the Newton increment, and update the three primary unknowns:

```
J   = eq.jac{1}; % Jacobian
res = eq.val;    % residual
upd = -(J \ res); % Newton update

% Update variables
p_ad.val   = p_ad.val   + upd(pIx);
bhp_ad.val = bhp_ad.val + upd(bhpIx);
qS_ad.val  = qS_ad.val  + upd(qSIx);
```



The sparsity pattern of the Jacobian is shown in the plot to the left of the code for the Newton update. The use of a two-point scheme on a 3D Cartesian grid will give a Jacobi matrix that has a heptadiagonal structure, except for the

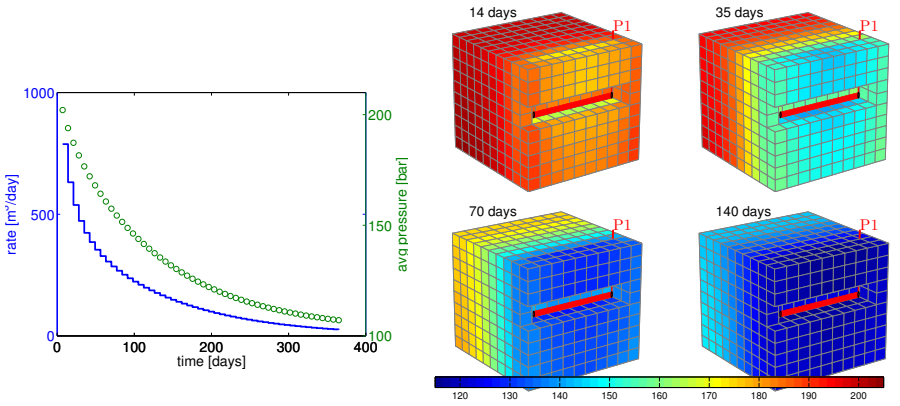


Fig. 7.4. Time evolution of the pressure solution for the compressible single-phase problem. The plot to the left shows the well rate (blue line) and average reservoir pressure (green circles) as function of time, and the plots to the right shows the pressure after two, five, ten, and twenty pressure steps.

off-diagonal entries in the two red rectangles that arise from the well equation and correspond to derivatives of this equation with respect to cell pressures.

Figure 7.4 shows a plot of the dynamics of the solution. Initially, the pressure is in hydrostatic equilibrium as shown in Figure 7.3. As the well starts to drain the reservoir, there will be a draw-down in the pressure near the well which will gradually drop from the well and outward. As a result, the average pressure inside the reservoir will be reduced, which again will cause a decay in the production rate.

7.5 Rapid prototyping

One particular advantage of using automatic differentiation in combination with the discrete differential and averaging operators is that it simplifies the testing of new models and alternative computational approaches. In this section, we will discuss two examples that hopefully demonstrates this aspect.

7.5.1 Pressure-dependent viscosity

In the model discussed in the previous section, the viscosity was assumed to be constant. However, in the general case the viscosity will increase with increasing pressures, and this effect may be significant for the high pressures seen inside a reservoir. To model this effect, we will introduce a linear dependence, rather than the exponential pressure-dependence used for the pore volume (7.6) and the fluid density (7.7). That is, we assume that the viscosity is given by

$$\mu(p) = \mu_0\big[1 + c_\mu(p - p_r)\big] \tag{7.13}$$

Having a pressure dependence means that we will have to change two parts of our discretization: the approximation of the Darcy flux across a cell face (7.8) and the flow rate through a well connection (7.12). Starting with the latter, we evaluate the viscosity using the same pressure as was used to evaluate the density, i.e.,

$$\boldsymbol{q}_c[w] = \frac{\rho(\boldsymbol{p}[N_c(w)])}{\mu(\boldsymbol{p}[N_c(w)])}\,\mathrm{WI}[w]\,\big(\boldsymbol{p}_c[w] - \boldsymbol{p}[N_c(w)]\big). \tag{7.14}$$

For the Darcy flux (7.8), we have two choices: either use a simple arithmetic average as in (7.9) to approximate the viscosity at each cell face, or .

$$\boldsymbol{v}[f] = -\frac{\boldsymbol{T}[f]}{\boldsymbol{\mu}_a[f]}\big(\mathtt{grad}(\boldsymbol{p}) - g\,\boldsymbol{\rho}_a[f]\,\mathtt{grad}(\boldsymbol{z})\big), \tag{7.15}$$

or replace the quotient of the transmissibility and the face viscosity by the harmonic average of the mobility $\lambda = \mathsf{K}/\mu$ in the adjacent cells. Both choices will lead to changes in the structure of the discrete nonlinear system, but because we are using automatic differentiation, all we have to do is code the formulas (7.13) to (7.17). Let us look at the details of the implementation in MRST, starting with the arithmetic approach.

Arithmetic average

First, we introduce a new anonymous function to evaluate the relation between viscosity and pressure:

```
[mu0,c_mu] = deal(5*centi*poise, 2e-3/barsa);
mu = @(p) mu0*(1+c_mu*(p-p_r));
```

Then we can replace the definition of the Darcy flux

```
v  = @(p)  -(T./mu(avg(p))).*( grad(p) - g*avg(rho(p)).*gradz );
```

and similarly for flow rate through each well connection

```
q_conn = @(p,bhp) WI.*(rho(p(wc))./ mu(p(wc))) .* (p_conn(bhp) - p(wc));
```

In Figure 7.5 we illustrate the effect of increasing the pressure dependence of the viscosity. Since the reference value is given at $p = 200$ bar, which is close to the initial pressure inside the reservoir, the more we increase c_μ, the lower μ will be in the pressure draw-down zone near the well. Therefore, we see a significantly higher initial production rate for $c_\mu = 0.005$ than for $c_\mu = 0$. On the other hand, the higher value of c_μ, the faster the draw-down effect of the well will propagate into the reservoir, inducing a reduction in reservoir pressure that will cause production to cease. In terms of overall production, a higher pressure dependence may be more advantageous as it will lead both to higher cumulative production early in the production period.

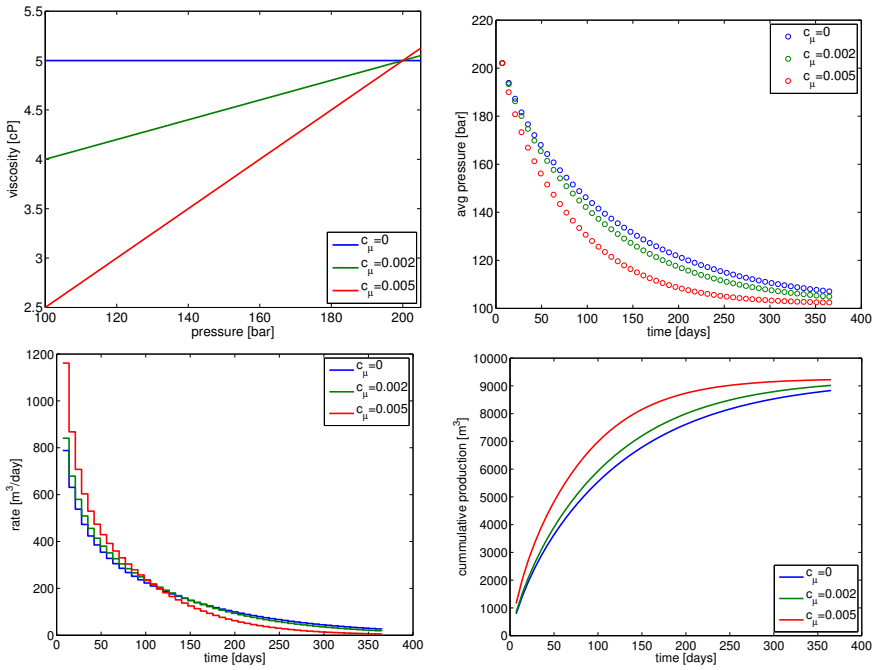


Fig. 7.5. The effect of increasing the degree of pressure-dependence for the viscosity.

Face mobility: harmonic average

A more correct approximation is to write Darcy's law based on mobility instead of using the quotient of the transmissibility and an averaged viscosity:

$$\boldsymbol{v}[f] = -\boldsymbol{\Lambda}[f]\left(\mathtt{grad}(\boldsymbol{p}) - g\,\boldsymbol{\rho}_a[f]\,\mathtt{grad}(\boldsymbol{z})\right). \tag{7.16}$$

The face mobility $\boldsymbol{\Lambda}[f]$ can be defined in the same way as the transmissibility is defined in terms of the half transmissibilities using harmonic averages. That is, if $\boldsymbol{T}[f,c]$ denotes the half transmissibility associated with face f and cell c, the face mobility $\boldsymbol{\Lambda}[f]$ for face f can be written as

$$\boldsymbol{\Lambda}[f] = \left(\frac{\boldsymbol{\mu}[N_1(f)]}{\boldsymbol{T}[f,N_1(f)]} + \frac{\boldsymbol{\mu}[N_2(f)]}{\boldsymbol{T}[f,N_2(f)]}\right)^{-1}. \tag{7.17}$$

In MRST, the corresponding code reads:

```
hf2cn = getCellNoFaces(G);
nhf = numel(hf2cn);
hf2f  = sparse(double(G.cells.faces(:,1)),(1:nhf)',1);
hf2if = hf2f(intInx,:);
hlam = @(mu,p) 1./(hf2if*(mu(p(hf2cn))./hT));

v  = @(p) -hlam(mu,p).*( grad(p) - g*avg(rho(p)).*gradz );
```

Here, `hf2cn` represents the maps N_1 and N_2 that enables us to sample the viscosity value in the correct cell for each half-face transmissibility, whereas `hf2if` represents a map from half faces (i.e., faces seen from a single cell) to global faces (which are shared by two cells). The map has a unit value in row i and column j if half face j belongs to global face i. Hence, premultiplying a vector of half-face quantities by `hf2if` amounts to summing the contributions from cells $N_1(f)$ and $N_2(f)$ for each global face f.

For a homogeneous model, using the harmonic average should produce simulation results that are identical (to machine precision) to those produced by using arithmetic average. With heterogeneous permeability, there will be small differences in the well rates and averaged pressures for the specific parameters considered herein. For sub-samples of the SPE 10 data set, we typically observe maximum relative differences in well rates of the order 10^{-3}.

7.5.2 Non-Newtonian fluid

Viscosity is the material property that measures a fluid's resistance to flow, i.e., the resistance to a change in shape, or to the movement of neighboring portions of the fluid relative to each other. The more viscous a fluid is, the less easily it will flow. In Newtonian fluids, the shear stress (the force applied per area tangential to the force), at any point is proportional to the strain rate (the symmetric part of the velocity gradient) at that point and the viscosity is the constant of proportionality. For non-Newtonian fluids, the relationship is no longer linear. The most common nonlinear behavior is shear thinning, in which the viscosity of the system decreases as the shear rate is increased. An example is paint, which should flow easily when leaving the brush, but stay on the surface and not drip once it has been applied. The second type of nonlinearity is shear thickening, in which the viscosity increases with increasing shear rate. A common example is the mixture of cornstarch and water. If you search YouTube for "cornstarch pool" you can view several spectacular videos of pools filled with this mixture. When stress is applied to the liquid, it exhibits properties like a solid and you may be able to run across its surface. However, if you go too slow, it will behave more like a liquid and you will fall in.

Solutions of large polymeric molecules are another example of shear-thinning liquids. In enhanced oil recovery, polymer solutions may be injected into reservoirs to improve unfavorable mobility ratios between oil and water and improve the sweep efficiency of the injected fluid. At low flow rates, the polymer molecule chains will tumble around randomly and present large resistance to flow. When the flow velocity increases, the viscosity will decrease as the molecules will gradually align themselves in the direction of increasing shear rate. A model of the rheology is given by

$$\mu = \mu_\infty + (\mu_0 - \mu_\infty)\left(1 + \left(\frac{K_c}{\mu_0}\right)^{\frac{2}{n-1}} \dot{\gamma}^2\right)^{\frac{n-1}{2}}, \tag{7.18}$$

where μ_0 represents the Newtonian viscosity at zero shear rate, μ_∞ represents the Newtonian viscosity at infinite shear rate, K_c represents the consistency index, and n represents the power-law exponent ($n < 1$). The shear rate $\dot{\gamma}$ in a porous medium can be approximated by

$$\dot{\gamma}_{\text{app}} = 6 \left(\frac{3n+1}{4n} \right)^{\frac{n}{n-1}} \frac{|\vec{v}|}{\sqrt{\mathsf{K}\phi}}. \tag{7.19}$$

Combining (7.18) and (7.19), we can write our model for the viscosity as

$$\mu = \mu_0 \left(1 + \bar{K}_c \frac{|\vec{v}|^2}{\mathsf{K}\phi} \right)^{\frac{n-1}{2}}, \quad \bar{K}_c = 36 \left(\frac{K_c}{\mu_0} \right)^{\frac{2}{n-1}} \left(\frac{3n+1}{4n} \right)^{\frac{2n}{n-1}}, \tag{7.20}$$

where we for simplicity have assumed that $\mu_\infty = 0$. In the following, we will show how easy it is to extend the simulator developed in the previous sections to model this non-Newtonian fluid behavior (see `nonNewtonianCell.m`). To simulate injection, we increase the bottom-hole pressure to 300 bar. Our rheology model has parameters:

```
mu0  = 100*centi*poise;
nmu  = 0.3;
Kc   = .1;
Kbc  = (Kc/mu0)^(2/(nmu-1))*36*((3*nmu+1)/(4*nmu))^(2*nmu/(nmu-1));
```

In principle, we could continue to solve the system using the same primary unknowns as before. However, it has proved convenient to write (7.20) in the form $\mu = \eta\,\mu_0$ and introduce η as an additional unknown. In each Newton step, we start by solving the equation for the shear factor η exactly for the given pressure distribution. This is done by initializing an AD variable for η, but not for p in `etaEq` so that this residual now only has one unknown, η. This will take out the implicit nature of Darcy's law and hence reduce the nonlinearity and simplify the solution of the global system.

```
while (resNorm > tol) && (nit < maxits)

  % Newton loop for eta (shear multiplier)
  [resNorm2,nit2] = deal(1e99, 0);
  eta_ad2 = initVariablesADI(eta_ad.val);
  while (resNorm2 > tol) && (nit2 <= maxits)
    eeq = etaEq(p_ad.val, eta_ad2);
    res = eeq.val;
    eta_ad2.val = eta_ad2.val - (eeq.jac{1} \ res);
    resNorm2 = norm(res);
    nit2     = nit2+1;
  end
  eta_ad.val = eta_ad2.val;
```

Once the shear factor has been computed for the values in the previous iterate, we can use the same approach as earlier to compute a Newton update for the full system. (Here, `etaEq` is treated as a system with two unknowns, p and η.)

```
eq1     = presEq(p_ad, p0, eta_ad, dt);
eq1(wc) = eq1(wc) - q_conn(p_ad, eta_ad, bhp_ad);
eqs = {eq1, etaEq(p_ad, eta_ad), ...
  rateEq(p_ad, eta_ad, bhp_ad, qS_ad), ctrlEq(bhp_ad)};
eq  = cat(eqs{:});
upd = -(eq.jac{1} \ eq.val); % Newton update
```

To finish the solver, we will of course need to define the flow equations and the extra equation for the shear multiplier. The main question to this end is: how should we compute $|\vec{v}|$? One solution could be to define $|\vec{v}|$ on each face as the flux divided by the face area. In other words, use a code like

```
phiK   = avg(rock.perm.*rock.poro)./G.faces.areas(intInx).^2;
v      = @(p, eta)   -(T./(mu0*eta)).*( grad(p) - g*avg(rho(p)).*gradz );
etaEq  = @(p, eta)   eta - (1 + Kbc*v(p,eta).^2./phiK).^((nmu-1)/2);
```

Although simple, this approach has three potential issues: First, it does not tell us how to compute the shear factor for the well perforations. Second, it disregards contributions from any tangential components of the velocity field. Third, the number of unknowns in the linear system will increase by almost a factor six since we now will have one extra unknown per internal face. The first issue is easy to fix: to get a representative value in the well cells, we simply average the η values from the cells' faces. If we now recall how the discrete divergence operator was defined, we realize that this operation is almost implemeted for us already: if `div(x)=-C'*x` computes the discrete divergence in each cell of the field `x` defined at the faces, then `cavg(x)=1/6*abs(C)'*x` will compute the average of `x` for each cell. In other words, our well equation becomes:

```
wavg   = @(eta) 1/6*abs(C(:,W.cells))'*eta;
q_conn = @(p, eta, bhp) ...
  WI .* (rho(p(wc)) ./ (mu0*wavg(eta))) .* (p_conn(bhp) - p(wc));
```

The second issue would have to be investigated in more detail and this is not within the scope of this book. The third issue is simply a disadvantage.

To get a method that consumes less memory, we can compute one η value per cell. Using the following formula, we can compute an approximate velocity $\vec{v}_i$ at the center of cell i

$$\vec{v}_i = \sum_{j \in N(i)} \frac{v_{ij}}{V_i} \left(\vec{c}_{ij} - \vec{c}_i \right), \tag{7.21}$$

where $N(i)$ is the map from cell i to its neighboring cells, v_{ij} is the flux between cell i and cell j, $\vec{c}_{ij}$ is the centroid of the corresponding face, and $\vec{c}_i$

is the centroid of cell i. For a Cartesian grid, this formula simplifies so that an approximate velocity can be obtained as the sum of the absolute value of the flux divided by the face area over all faces that make up a cell. Using a similar trick as we used to compute η in well cells above, our implementation follows trivially. We first define the averaging operator to compute cell velocity

```
aC = bsxfun(@rdivide, 0.5*abs(C), G.faces.areas(intInx))';
cavg = @(x) aC*x;
```

In doing so, we also rename our old averaging operator `avg` as `favg` to avoid confusion and make it more clear that this operator maps from cell values to face values. Then we can define the needed equations:

```
phiK  = rock.perm.*rock.poro;
gradz = grad(G.cells.centroids(:,3));
v = @(p, eta)
   -(T./(mu0*favg(eta))).*( grad(p) - g*favg(rho(p)).*gradz );
etaEq = @(p, eta)
   eta - ( 1 + Kbc* cavg(v(p,eta)).^2 ./phiK ).^((nmu-1)/2);
presEq= @(p, p0, eta, dt)   ...
   (1/dt)*(pv(p).*rho(p) - pv(p0).*rho(p0)) + div(favg(rho(p)).*v(p, eta));
```

With this approach the well equation becomes particularly simple since all we need to do is to sample the η value from the correct cell:

```
q_conn = @(p, eta, bhp) ...
    WI .* (rho(p(wc)) ./ (mu0*eta(wc))) .* (p_conn(bhp) - p(wc));
```

A potential drawback of this second approach is that it may introduce numerical smearing, but this will, on the other hand, most likely increase the robustness of the resulting scheme.

In Figure 7.6 we compare the predicted flow rates and average reservoir pressure for two different fluid models: one that assumes that the fluid is a standard Newtonian fluid (i.e., $\eta \equiv 1$) and one that models shear thinning, which has been computed by both methods discussed above. With shear thinning, the higher pressure in the injection well will cause a decrease in the viscosity which will lead to significantly higher injection rates than for the Newtonian fluid and hence a higher average reservoir pressure. Perhaps more interesting is the large discrepancy in the rates and pressures predicted by the face-based and the cell-based simulation algorithms. If we in the face-based method disregard the shear multiplier `q_conn`, the predicted rate and pressure build-up is smaller than what is predicted by the cell-based method and closer to the Newtonian fluid case. We take this as evidence that the differences between the cell and the face-base methods to a large extent can be explained by differences in the discretized well models and their ability to capture the formation and propagation of the strong initial transient. To further back this up, we have included results from a simulation with ten times as many time steps in Figure 7.7, which also includes plots of the evolution of $\min(\eta)$ as a

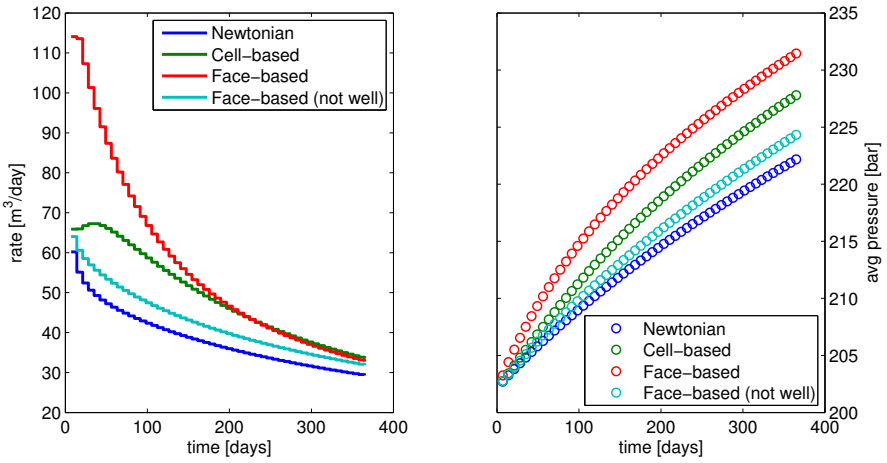


Fig. 7.6. Single-phase injection of a highly viscous, shear-thinning fluid computed by four different simulation methods: (i) fluid assumed to be Newtonian, (ii) shear multiplier η computed in cells, (iii) shear multiplier computed at faces, and (iv) shear multiplier computed at faces, but $\eta \equiv 1$ used in well model.

function of time. Whereas the face-based method predicts a large, immediate drop in viscosity in the near-well region, the viscosity drop predicted by the cell-based method is much smaller during the first 20–30 days. This will result in a delay in the peak in the injection rate and a much smaller injected volume.

We will leave the discussion here. The parameters used in the example were chosen quite haphazardly to demonstrate a pronounced shear-thinning effect. Which method is the most correct for real computations, is a question that goes beyond the current scope, and could probably best be answered by verifying against observed data for a real case. Our point here, was mainly to demonstrate the capability of rapid prototyping that comes with the use of MRST. However, as the example shows, this lunch is not completely free: you will still have to understand features and limitations of the models and discretizations you choose to prototype.

Computer exercises:

1. Apply the compressible pressure solver from Section 7.4 to the quarter five-spot problem discussed in Section 6.4.1.
2. Apply the compressible pressure solver from Section 7.4 to the three different grid models studied in Section 6.4.3 that were derived from the `seamount` data set. Replace the fixed boundary conditions by a no-flow condition.
3. Investigate the claim on page 178 that the difference between using a arithmetic average of the viscosity and a harmonic average of the fluid mobility

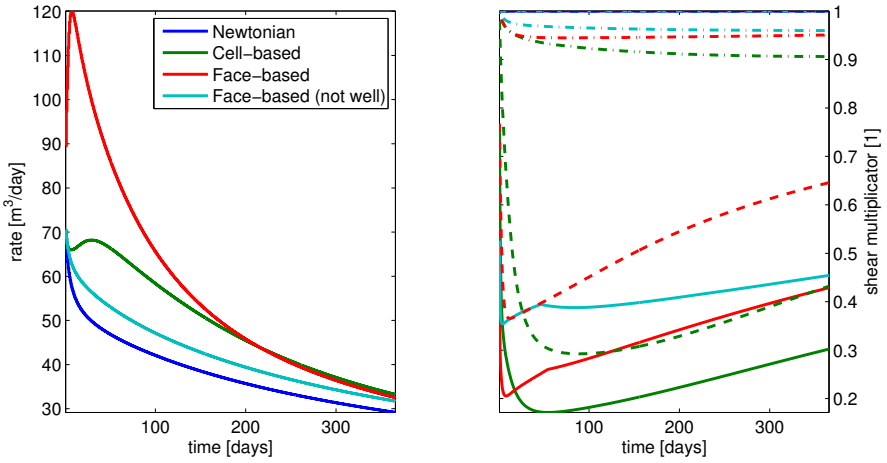


Fig. 7.7. Single-phase injection of a highly viscous, shear-thinning fluid; simulation with $\Delta t = 1/520$ year. The right plot shows the evolution of η as a function of time: solid lines show $\min(\eta)$ over all cells, dashed lines $\min(\eta)$ over the perforated cells, and dash-dotted lines average η value.

is typically small. To this end, you can for instance use the following subsample from the SPE10 data set:

```
rock = SPE10_rock(41:50,101:110,1:10);
rock.perm = rock.perm*milli*darcy;
```

4. Extend the compressible solver in Section 7.4 to incorporate other boundary conditions than no flow.
5. Use the implementation introduced in Section 7.4 as a template to develop a solver for slightly compressible flow as discussed on page 114 in Section 5.2. How large can c_f be before the assumptions in the slightly compressible model become inaccurate? Use different heterogeneities, well placements, and/or model geometries to investigate this question in more detail.
6. Try to compute time-of-flight for the compressible example from Section 7.4 by extending the equation set to also include the time-of-flight equation (5.37). Hint: the time-of-flight and the pressure equations need not be solved as a coupled system.
7. Same as above, except that you should try to reuse the solver introduced in Section 6.3. Hint: you must first reconstruct fluxes from the computed pressure and then construct a state object to communicate with the TOF solver.
8. The non-Newtonian fluid example in Section 7.5.2 will have a strong transient during the first 30–100 days. Try to implement adaptive time steps that utilizes this fact. Can you come up with a strategy that automatically chooses good time steps?

8

Consistent Discretizations on Polyhedral Grids

In the previous chapters, we have used the two-point flux-approximation (TPFA) scheme introduced in Section 5.4.1 to discretize the elliptic Laplace operator $\mathcal{L} = \nabla \cdot \mathsf{K}\nabla$ in space. That is, if p_i and p_k denote the average pressures in two neighboring cells Ω_i and Ω_k, then the flux across the interface Γ_{ik} between them is given as

$$v_{ik} = T_{ik}(p_i - p_k), \tag{8.1}$$

where the transmissibility T_{ik} depends on the geometry of the two cells and the associated permeability tensors K_i and K_k. The TPFA scheme is very robust and widely used both in academia and industry. However, the method is only consistent for certain combinations of grids and permeability tensors K and hence will not generally be convergent. To see this, we assume that K is a homogeneous symmetric tensor

$$\mathsf{K} = \begin{bmatrix} \mathsf{K}_{xx} & \mathsf{K}_{xy} \\ \mathsf{K}_{xy} & \mathsf{K}_{yy} \end{bmatrix}.$$

Consider now the flux across an interface Γ_{ik} between two cells Ω_i and Ω_k in a 2D Cartesian grid, whose normal vector $\vec{n}_{i,k}$ points in the x-direction so that $\vec{n}_{i,k} = \vec{c}_{i,k} = (1, 0)$; see Figure 8.1. Using Darcy's law gives that $\vec{v} \cdot \vec{n} = -(\mathsf{K}_{xx}\partial_x p + \mathsf{K}_{xy}\partial_y p)$, from which it follows that the flux across the interface will have two components, one orthogonal and one transverse,

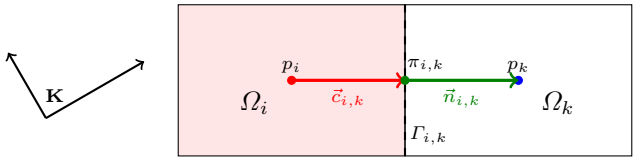


Fig. 8.1. Two cells in a Cartesian grid and a full permeability tensor whose principal axes are not aligned with the coordinate system.

$$v_{ik} = -\int_{\Gamma_{ik}} \mathsf{K}\nabla p \cdot \vec{n}\, ds = -\int_{\Gamma_{ik}} \left(\mathsf{K}_{xx}\,\partial_x p + \mathsf{K}_{xy}\,\partial_y p\right) ds$$

that correspond to the x-derivative and the y-derivative of the pressure, respectively. In the two-point approximation (8.1), the only points we can use to approximate these flux contributions are the pressures p_i and p_k, but since these two pressures are associated with the same y-value, their difference can only be used to estimate $\partial_x p$ and not $\partial_y p$. This means that the TPFA method cannot account for the transverse flux contribution $\mathsf{K}_{xy}\partial_y p$ and will hence generally not be consistent. The only exception is if $\mathsf{K}_{xy} \equiv 0$ in all cells in the grid, in which case there are no fluxes in the transverse direction. To link the above derivation to the geometry of the cells, we can use the one-sided definition of fluxes (5.48) of the two-point scheme, two write

$$v_{i,k} \approx T_{i,k}(p_i - \pi_{i,k}) = \frac{A_{i,k}}{|\vec{c}_{i,k}|^2}\, \mathsf{K}\vec{c}_{i,k} \cdot \vec{n}_{i,k}(p_i - \pi_{i,k}).$$

More generally, we have that *the TPFA scheme is only convergent for K-orthogonal grids*. An example of a K-orthogonal grid is a grid in which all cells are parallelepipeds in 3D or parallelograms in 2D and satisfy the condition

$$\vec{n}_{i,j} \cdot \mathsf{K}\vec{n}_{i,k} = 0, \qquad \forall j \neq k, \tag{8.2}$$

where $\vec{n}_{i,j}$ and $\vec{n}_{i,k}$ denote normal vectors to faces of cell number i.

What about grids that are not parallelepipeds or parallelograms? For simplicity, we only consider the 2D case. Let $\vec{c} = (c_1, c_2)$ denote the vector from the center of cell Ω_i to the centroid of face Γ_{ik}, see Figure 5.4. If we let $\vec{c}_\perp = (c_2, -c_1)$ denote a vector that is orthogonal to $\vec{c}$, then any constant pressure gradient inside Ω_i can be written as $\nabla p = p_1\vec{c} + p_2\vec{c}_\perp$, where p_1 can be determined from p_i and $\pi_{i,k}$ and p_2 is some unknown constant. Inserted into the definition of the face flux, this gives

$$v_{i,k} = -\int_{\Gamma_{ik}} \left(p_1 \mathsf{K}\vec{c} \cdot \vec{n} + p_2 \mathsf{K}\vec{c}_\perp \cdot \vec{n}\right) ds.$$

For the two-point scheme to be correct, the second term in the integrand must be zero. Setting $\vec{n} = (n_1, n_2)$, we have that

$$0 = \mathsf{K}\vec{c}_\perp \cdot \vec{n} = (K_1 c_2, -K_2 c_1) \cdot (n_1, n_2) = (K_1 n_1 c_2 - K_2 n_2 c_1) = (\mathsf{K}\vec{n}) \times \vec{c}.$$

In other words, a sufficient condition for a polygonal grid to be K-orthogonal is that $(\mathsf{K}\vec{n}_{i,k}) \parallel \vec{c}_{i,k}$ for all cells in the grid, in which case the transverse flux contributions are all zero and hence need not be estimated. Consequently, the TPFA method will be consistent.

The lack of consistency in the TPFA scheme for grids that are not K-orthogonal may lead to significant *grid-orientation effects*, i.e., artifacts or errors in the solution that will appear in varying degree depending upon the

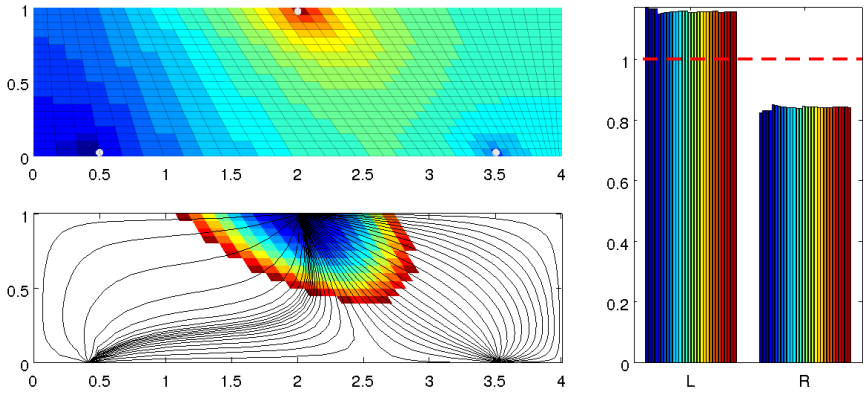


Fig. 8.2. Solution of a symmetric flow problem in a homogeneous domain using the TPFA method on a skew grid that is not K-orthogonal. The upper-left plot shows the pressure distribution, whereas the lower-left plot shows streamlines and time-of-flight values less than 0.25 PVI. The plot to the right shows time-of-flight in left (L) and right (R) sink for a series of refined grids of dimension $(20n + 1) \times 10n$ for $n = 1, \ldots, 30$. The solutions do not converge toward the analytical solution shown as the dashed red line.

angles the cell faces make with the principal directions of the permeability tensor. We have already seen an example of grid effects in Figure 6.6 in Section 6.4.3, where the use of a triangular grids caused large deviations from what should have been an almost symmetric pressure draw-down. To demonstrate grid-orientation effects and lack of convergence more clearly, we look at a simple example that has been widely used to teach reservoir engineers the importance of aligning the grid with the principal permeability directions.

Consider a homogeneous reservoir in the form of a 2D rectangle $[0, 4] \times [0, 1]$. Flow is driven inside the reservoir by a fluid source with unit rate located at $(1, 0.975)$, and two fluid sinks located at $(0.5, 0.025)$ and $(3.5, 0.025)$, each having a rate equal one half. The domain is discretized by a grid that is uniform along the north side and graded along the south side:

```
G = cartGrid([41,20],[2,1]);
makeSkew = @(c) c(:,1) + .4*(1-(c(:,1)-1).^2).*(1-c(:,2));
G.nodes.coords(:,1) = 2*makeSkew(G.nodes.coords);
```

The grid is not K-orthogonal, and we therefore cannot expect that the TPFA method will converge. Complete setup of the problem can be found in the script `1phase/showInconsistentTPFA.m`.

Figure 8.2 reports the result of a convergence study. All the computed solutions exhibit the same lack of symmetry that is seen in the solution on the 41×20 grid. Moreover, the large difference in travel times from the injector to the two producers does not decay with increasing grid resolution, which confirms the expected lack of convergence for an inconsistent scheme.

8.1 The Mixed Finite-Element Method

There are several ways to formulate consistent discretization methods. In this section, we will introduce a method, the mixed finite-element method [15], which is based on a formulation that is quite different from the one we have seen so far for the inconsistent two-point method. The first new idea is that instead of forming a second-order elliptic pressure equation, we will form a system of equations consisting of two first-order equations: the mass-conservation equation and Darcy's law. This means that unlike for the TPFA method, in which discrete fluxes are computed by post-processing the pressure solution, the mixed method solves for pressure and fluxes simultaneously. The second new idea is to look for solutions that satisfy the flow equations in a weak sense; that is, look for solutions that fulfill the equation when multiplied with a suitable test function and integrated in space. The third new idea is to express the unknown solution as a linear combination of a set of basis functions that take the form of piecewise polynomials with localized support.

The mixed finite-element method is not implemented directly in MRST, but the key ideas of this method will be instrumental in the development of a general class of finite-volume discretizations that will be outlined in Section 8.2 and discussed in more details in the remaining sections of the chapter. As a precursor to this discussion, we will in the following present the mixed finite-element method in some detail.

8.1.1 Continuous Formulation

We start by restating the continuous flow equation in full detail:

$$\nabla \cdot \vec{v} = q, \qquad \vec{v} = -\mathsf{K}\nabla p, \qquad \vec{x} \in \Omega \subset \mathbb{R}^d \tag{8.3}$$

with boundary conditions $\vec{v} \times \vec{n}$ for $\vec{x} \in \partial\Omega$. For compatibility, we require that $\int_\Omega q \, d\vec{x} = 0$. Since this is a pure Neumann boundary-value problem, the pressure p is only defined up to an arbitrary constant, and as an extra constraint, we require that $\int_\Omega p \, d\vec{x} = 0$.

In the mixed method, we will look for solutions that lie in an abstract function space. To this end, we need two spaces: $L^2(\Omega)$ is the space of square integrable functions, and H_0^{div} is a so-called Sobolev space defined as

$$H_0^{\mathrm{div}}(\Omega) = \{\vec{v} \in L^2(\Omega)^d : \nabla \cdot \vec{v} \in L^2(\Omega) \text{ and } \vec{v} \cdot \vec{n} \text{ on } \partial\Omega\}, \tag{8.4}$$

i.e., the set of square-integrable, vector-valued functions that have compact support in Ω and whose divergence is also square integrable. The mixed formulation of (8.3) now reads: find a pair $(p, \vec{v})$ that lies in $L^2(\Omega) \times H_0^{\mathrm{div}}(\Omega)$ and satisfies

$$\begin{aligned} \int_\Omega \vec{u} \cdot \mathsf{K}^{-1}\vec{v} \, d\vec{x} - \int_\Omega p \, \nabla \cdot \vec{u} \, d\vec{x} = 0, \qquad & \forall \vec{u} \in H_0^{\mathrm{div}}(\Omega), \\ \int_\Omega w \, \nabla \cdot \vec{v} \, d\vec{x} = \int_\Omega q w \, d\vec{x}, \qquad & \forall w \in L^2(\Omega). \end{aligned} \tag{8.5}$$

The first equation follows by multiplying Darcy's law by K^{-1} applying the divergence theorem to the term involving the pressure gradient. (Recall that $\vec{u}$ is zero on $\partial\Omega$ by definition so that the term $\int_{\partial\Omega} p\vec{u}\cdot\vec{n}\,ds = 0$).

Equation (8.5) can be written on a more compact form if we introduce the following three inner products

$$\begin{aligned} b(\cdot,\cdot) &: H_0^{\mathrm{div}} \times H_0^{\mathrm{div}} \to \mathbb{R}, & b(\vec{u},\vec{v}) &= \int_\Omega \vec{u}\cdot\mathsf{K}^{-1}\vec{v}\,d\vec{x} \\ c(\cdot,\cdot) &: H_0^{\mathrm{div}} \times L^2 \to \mathbb{R}, & c(\vec{u},p) &= \int_\Omega p\,\nabla\cdot\vec{u}\,d\vec{x} \\ (\cdot,\cdot) &: L^2 \times L^2 \to \mathbb{R}, & (q,w) &= \int_\Omega qw\,d\vec{x}, \end{aligned} \tag{8.6}$$

and write

$$\begin{aligned} b(\vec{u},\vec{v}) - c(\vec{u},p) &= 0 \\ c(\vec{v},w) &= (q,w) \end{aligned} \tag{8.7}$$

The mixed formulation (8.7) can alternatively be derived by minimizing the energy functional

$$I(\vec{v}) = \frac{1}{2}\int_\Omega \vec{v}\cdot\mathsf{K}^{-1}\vec{v}\,d\vec{x},$$

over all functions $\vec{v} \in H_0^{\mathrm{div}}(\Omega)$ that are mass conservative, i.e., subject to the constraint

$$\nabla\cdot\vec{v} = q.$$

The common strategy for solving such mimization problems is to introduce a Lagrangian functional, which in our case reads

$$L(\vec{v},p) = \int_\Omega \Big(\tfrac{1}{2}\vec{v}\cdot\mathsf{K}^{-1}\vec{v} - p\big(\nabla\cdot\vec{v} - q\big)\Big)\,d\vec{x} = \frac{1}{2}b(\vec{v},\vec{v}) - c(\vec{v},p) + (p,q),$$

where p is the so-called the Lagrangian multiplier. At a minimum of L, we must have that $\partial L/\partial\vec{v} = 0$. Looking at an increment $\vec{u}$ of $\vec{v}$, we have the requirement that

$$0 = \frac{\partial L}{\partial\vec{v}}\vec{u} = \lim_{\varepsilon\to 0}\frac{1}{\varepsilon}\big[L(\vec{v}+\varepsilon\vec{u},p) - L(\vec{v},p)\big] = b(\vec{u},\vec{v}) - c(\vec{u},p)$$

must be zero for all $u \in H_0^{\mathrm{div}}(\Omega)$. Similarly, we can show that

$$0 = \frac{\partial L}{\partial p}w = \lim_{\varepsilon\to 0}\frac{1}{\varepsilon}\big[L(\vec{v},p+\varepsilon w) - L(\vec{v},p)\big] = -c(\vec{v},w) + (q,w)$$

must be zero for all $w \in L^2(\Omega)$. To show that the solution $\vec{v}$ is a minimal point, we consider a perturbation $\vec{v}+\vec{u}$ that satisfies the constraints, i.e., $\vec{u} \in H_0^{\mathrm{div}}$ and $\nabla\cdot\vec{u} = 0$. For the energy functional we have that

$$I(\vec{v}+\vec{u}) = \frac{1}{2}\int_\Omega (\vec{v}+\vec{u})\cdot \mathsf{K}^{-1}(\vec{v}+\vec{u})\,d\vec{x} = I(\vec{v}) + I(\vec{u}) + b(\vec{u},\vec{v})$$
$$= I(\vec{v}) + I(\vec{u}) + c(\vec{u},p) = I(\vec{v}) + I(\vec{u}) > I(\vec{v}),$$

which proves that $\vec{v}$ is indeed a minimum of I. One can also prove that the solution $(p,\vec{v})$ is a saddle-point of the Lagrange functional, i.e., that $L(\vec{v},w) \le L(\vec{v},p) \le L(\vec{u},p)$ for all $\vec{u}\neq\vec{v}$ and $w \neq p$. The right inequality can be shown as follows

$$L(\vec{v}+\vec{u},p) = \tfrac{1}{2}b(\vec{v}+\vec{u},\vec{v}+\vec{u}) - c(\vec{v}+\vec{u},p) + (q,p)$$
$$= L(\vec{v},p) + I(\vec{u}) + b(\vec{u},\vec{v}) - c(\vec{u},p) = L(\vec{v},p) + I(\vec{u}) > L(\vec{v},p).$$

The left inequality follows in a similar manner.

8.1.2 Discrete Formulation

To discretize the mixed formulation, (8.5), we introduce a grid $\Omega_h = \cup\Omega_i$ and replace $L^2(\Omega)$ and $H_0^{\mathrm{div}}(\Omega)$ by finite-dimensional subspaces U and V defined over the grid. Likewise, we rewrite the inner products (8.6) as sums of integrals localized to cells

$$\begin{aligned} b(\cdot,\cdot)_h &: V\times V\to\mathbb{R}, \qquad & b(\vec{u},\vec{v})_h &= \sum_i \int_{\Omega_i} \vec{u}\cdot\mathsf{K}^{-1}\vec{v}\,d\vec{x} \\ c(\cdot,\cdot)_h &: V\times U\to\mathbb{R}, \qquad & c(\vec{u},p)_h &= \sum_i \int_{\Omega_i} p\,\nabla\cdot\vec{u}\,d\vec{x} \\ (\cdot,\cdot)_h &: U\times U\to\mathbb{R}, \qquad & (q,w)_h &= \sum_i \int_{\Omega_i} qw\,d\vec{x}, \end{aligned} \tag{8.8}$$

To obtain a practical numerical method, the spaces U and V are typically defined as piecewise polynomial functions that are nonzero on a small collection of grid cells. For instance, in the Raviart–Thomas method [55, 15] of lowest order for triangular, tetrahedral or regular parallelepiped grids, $L^2(\Omega)$ is replaced by

$$U = \{p\in L^2(\cup\Omega_i) : p|_{\Omega_i} \text{ is constant } \forall\Omega_i\subset\Omega\} = \mathrm{span}\{\chi_i\},$$

where χ_i is the characteristic function of grid cell Ω_i,

$$\chi_i = \begin{cases} 1, & \vec{x}\in\Omega_i \\ 0, & \text{otherwise.} \end{cases}$$

Likewise, $H_0^{\mathrm{div}}(\Omega)$ is replaced by a space V consisting of functions $\vec{v}\in H_0^{\mathrm{div}}(\cup\Omega_i)$ that have linear components on each grid cell $\Omega_i\in\Omega$, has normal components $\vec{v}\cdot\vec{n}_{ik}$ that are constant on each cell interface Γ_{ik} and continuous across these interfaces. These requirements are satisfied by functions

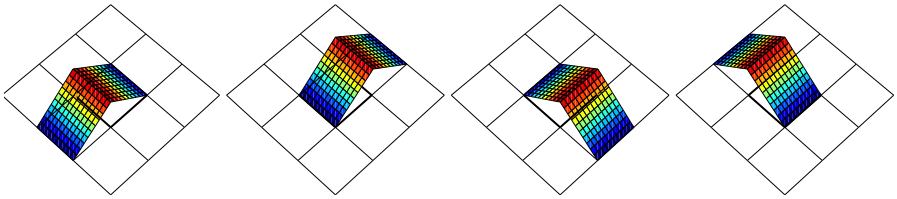


Fig. 8.3. Illustration of the velocity basis for the lowest-order Raviart–Thomas (RT0) method for a 2D Cartesian grid. There are four basis functions that have nonzero support in the interior of the center cell. The basis functions correspond to degrees-of-freedom associated with the west, east, south, and north cell faces.

$\vec{v} \in P_1(\Omega_i)^d$, i.e., first order polynomials that on each grid cell takes the form $\vec{v}(\vec{x}) = \vec{a} + b\vec{x}$, where $\vec{a}$ and b are vector-valued and scalar constants, respectively. These functions can be parametrized in terms of the faces between two neighboring cells. This means that we can write $V = \text{span}\{\vec{\psi}_{ik}\}$, where each function $\vec{\psi}_{ik}$ is defined as

$$\vec{\psi}_{ik} \in \mathcal{P}_1(\Omega_i)^d \cup \mathcal{P}_1(\Omega_k)^d \quad \text{and} \quad (\vec{\psi}_{ik} \cdot \vec{n}_{jl})|_{\Gamma_{jl}} = \begin{cases} 1 & \text{if} \quad \Gamma_{jl} = \Gamma_{ik}, \\ 0 & \text{otherwise}, \end{cases}$$

Figure 8.3 illustrates the four nonzero basis functions that result in the special case of a Cartesian grid in 2D.

To derive a fully discrete method, we use χ_i and $\vec{\psi}_{ik}$ as our trial functions and express the unknown $p(\vec{x})$ and $\vec{v}(\vec{x})$ as sums over these trial functions, $p = \sum_i p_i \chi_i$ and $\vec{v} = \sum_{ik} v_{ik} \vec{\psi}_{ik}$. The corresponding degrees-of-freedom, p_i associated with each cell and v_{ik} associated with each interface Γ_{ik}, are collected in two vectors $\boldsymbol{p} = \{p_i\}$ and $\boldsymbol{v} = \{v_{ik}\}$. Using the same functions as test functions, we derive a linear system of the form

$$\begin{bmatrix} \boldsymbol{B} & -\boldsymbol{C}^T \\ \boldsymbol{C} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{v} \\ \boldsymbol{p} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{q} \end{bmatrix}. \tag{8.9}$$

Here, $\boldsymbol{B} = [b_{ik,jl}]$, $\boldsymbol{C} = [c_{i,kl}]$, and $\boldsymbol{q} = [q_i]$, where:

$$\begin{aligned} b_{ik,jl} &= b(\vec{\psi}_{ik}, \vec{\psi}_{jl})_h = \sum_\ell \int_{\Omega_\ell} \vec{\psi}_{ik} \cdot \mathsf{K}^{-1} \vec{\psi}_{jl} \, d\vec{x}, \\ c_{i,kl} &= c(\vec{\psi}_{kl}, \chi_i)_h = \int_{\Omega_i} \nabla \cdot \vec{\psi}_{kl} \, d\vec{x}, \\ q_i &= (q, \chi_i)_h = \int_{\Omega_i} q \, d\vec{x}. \end{aligned} \tag{8.10}$$

Note that for the first-order Raviart–Thomas finite elements, we have

$$c_{i,kl} = \begin{cases} 1, & \text{if } i = k, \\ -1, & \text{if } i = l, \\ 0, & \text{otherwise.} \end{cases}$$

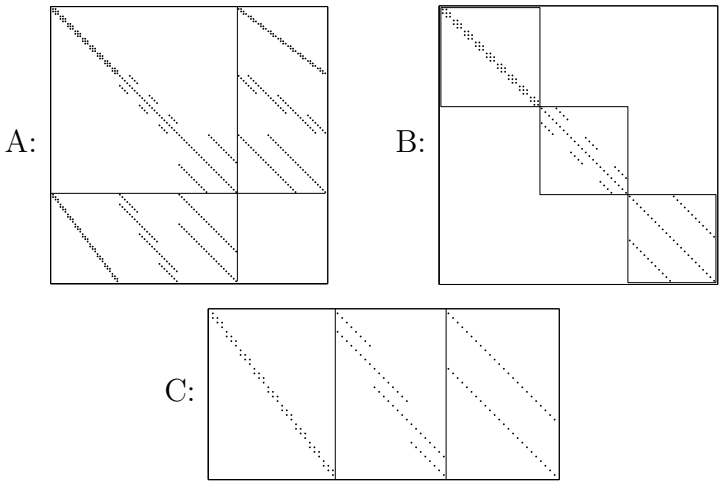


Fig. 8.4. Sparsity patterns for the full mixed finite-element matrix $\boldsymbol{A}$, and the matrix blocks $\boldsymbol{B}$ and $\boldsymbol{C}$ for a $4 \times 3 \times 3$ Cartesian grid.

The matrix entries $b_{ik,jl}$ depend on the geometry of the grid cells and whether K is isotropic or anisotropic. Unlike the two-point method, mixed methods are consistent and will therefore be convergent also on grids that are not K-orthogonal.

In [2], we presented a simple MATLAB code that in approximately seventy-five code lines implements the lowest-order Raviart–Thomas mixed finite-element method on regular hexahedral grids for flow problems with diagonal tensor permeability in two or three spatial dimensions. The code is divided into three parts: assembly of the $\boldsymbol{B}$ block, assembly of the $\mathbf{C}$ block, and a main routine that loads data (permeability and grid), assembles the whole matrix, and solves the system. Figure 8.4 illustrates the sparsity pattern of the mixed system for a $n_x \times n_y \times n_z$ Cartesian grid. The system matrix clearly has the block structure given in (8.9). The matrix blocks $\boldsymbol{B}$ and $\boldsymbol{C}$ each have three nonzero blocks that correspond to velocity basis functions oriented along the three axial directions. That is, the degrees-of-freedom at the interfaces (the fluxes) have been numbered in the same way as the grid cells; i.e., first faces orthogonal to the x-direction, then faces orthogonal to the y-direction, and finally faces orthogonal to the z-direction. This gives $\mathbf{B}$ a heptadiagonal structure.

To implement the code presented in [2], we exploited the simple geometry of the Cartesian grid to integrate the Raviart–Thomas basis functions analytically and perform a direct assembly of the matrix blocks to give a MATLAB code that was both compact and efficient. For more general grids, one would typically perform an elementwise assembly by looping over all cells in the grid, mapping each of them back to a reference element, on which the integration of basis functions is performed using some numerical quadrature rule.

In our experience, this procedure becomes cumbersome to implement for general stratigraphic grids and would typically require the use of several different reference elements. Because MRST is designed to work on general polyhedral grids in 3D, the software does not supply any direct implementation of mixed finite-element methods. Instead, the closest we get to a mixed method is a finite-volume method in which the half-transmissibilities are equivalent to the discrete inner products for the lowest-order Raviart–Thomas (RT0) method on rectangular cuboids. More details will be explained in Section 8.3.4. In the rest of this section, we will introduce an alternative formulation of the mixed method that gives a smaller linear system that is better conditioned.

8.1.3 Hybrid formulation

In the same way as we proved that the solution of the mixed problem is a saddle point, one can show that the linear system in (8.9) is indefinite. Indefinite systems are harder to solve and generally require special linear solvers. In the following, we therefore introduce an alternative formulation of the mixed method that, when discretized, will give a positive-definite discrete system and thereby simplify the computation of a discrete solution. Later in the chapter, this so-called hybrid formulation will form the basis for a general family of finite-volume discretizations on polygonal and polyhedral grids.

In a hybrid formulation, the need to solve a saddle-point problem is avoided by lifting the constraint that the normal velocity must be continuous across cell faces and instead integrate (8.3) to get a weak form that contains jump terms at the cell interfaces. Continuity of the normal velocity component is then reintroduced by adding an extra set of equations, in which the pressure π at the cell interfaces plays the role of Lagrange multipliers. (Recall how Lagrange multipliers were used to impose mass conservation as a constraint in the minimization procedure used to derive the mixed formulation in Section 8.1.1). Introducing Lagrange multipliers does not change $\vec{v}$ or p, but enables the recovery of pressure values at cell faces, in addition to introducing a change in the structure of the weak equations. Mathematically, mixed hybrid formulation reads: find $(\vec{v}, p, \pi) \in H_0^{\mathrm{div}}(\Omega_h) \times L^2(\Omega_h) \times H^{\frac{1}{2}}(\Gamma_h)$ such that

$$\begin{aligned} \sum_i \int_{\Omega_i} \left(\vec{u} \cdot \mathsf{K}^{-1} \vec{v} - p \nabla \cdot \vec{u} \right) d\vec{x} + \sum_{ik} \int_{\Gamma_{ik}} \pi \, \vec{u} \cdot \vec{n} \, ds = 0, \\ \sum_i \int_{\Omega_i} w \nabla \cdot \vec{v} \, d\vec{x} = \int_{\Omega_h} q w \, d\vec{x}, \\ \sum_{ik} \int_{\Gamma_{ik}} \mu \, \vec{v} \cdot \vec{n} \, ds = 0 \end{aligned} \tag{8.11}$$

for all test functions $\vec{u} \in H_0^{\mathrm{div}}(\Omega_h)$, $w \in L^2(\Omega_h)$, and $\mu \in H^{\frac{1}{2}}(\Gamma_h)$. Here, $\Gamma_h = \cup \Gamma_{ik} = \cup \partial\Omega_h \setminus \partial\Omega$ denotes all the interior faces of the grid and $H^{\frac{1}{2}}(\Gamma_h)$

is the space spanned by the traces[1] of functions in $H^1(\Omega_h)$, i.e., the space of square integrable functions whose derivatives are also square integrable. As for the mixed formulation, we can introduce inner products to write the weak equations (8.11) in a more compact form,

$$\begin{aligned} b(\vec{u},\vec{v}) - c(\vec{u},p) + d(\vec{u},\pi) &= 0 \\ c(\vec{v},w) &= (q,w) \\ d(\vec{v},\mu) &= 0, \end{aligned} \tag{8.12}$$

where the inner products $b(\cdot,\cdot)$, $c(\cdot,\cdot)$, and $(\cdot,\cdot)$ are defined as in (8.6), and $d(\cdot,\cdot)$ is a new inner product defined over the interior faces,

$$d(\cdot,\cdot)_h : H_0^{\mathrm{div}}(\Omega_h) \times H^{\frac{1}{2}}(\Gamma_h) \to \mathbb{R}, \qquad d(\vec{v},\pi) = \sum_{ik} \int_{\Gamma_{ik}} \pi\, \vec{v}\cdot\vec{n}\, ds. \tag{8.13}$$

To derive a fully discrete problem, we proceed in the exact same way as for the mixed problem by first replacing the function spaces L^2, H_0^{div}, and $H^{\frac{1}{2}}$ by finite-dimensional subspaces V and U that are spanned by piecewise polynomial basis functions with local support as discussed above. In the lowest-order approximation, the finite-dimensional space Π consists of functions that are constant on each face,

$$\Pi = \mathrm{span}\{\mu_{ik}\}, \qquad \mu_{ik}(\vec{x}) = \begin{cases} 1, & \text{if } \vec{x} \in \Gamma_{ik}, \\ 0, & \text{otherwise.} \end{cases} \tag{8.14}$$

Using these basis functions as test and trial functions, one can derive a discrete linear system of the form,

$$\begin{bmatrix} \boldsymbol{B} & \boldsymbol{C} & \boldsymbol{D} \\ \boldsymbol{C}^{\mathsf{T}} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{D}^{\mathsf{T}} & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{v} \\ -\boldsymbol{p} \\ \boldsymbol{\pi} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{q} \\ \boldsymbol{0} \end{bmatrix}, \tag{8.15}$$

where the vectors $\boldsymbol{v}$, $\boldsymbol{p}$, and $\boldsymbol{\pi}$ collect the degrees-of-freedom associated with fluxes across the cell interfaces, face pressures, and cell pressures, respectively, the matrix blocks $\boldsymbol{B}$ and $\boldsymbol{C}$ are defined as in (8.10), and $\boldsymbol{D}$ has two non-zero entries per column (one entry for each side of the cell interfaces).

The linear system (8.15) is an example of a sparse, symmetric, indefinite system (i.e., a system $\boldsymbol{A}$ whose quadratic form $\boldsymbol{x}^T\boldsymbol{A}\boldsymbol{x}$ takes both positive and negative values). Several methods for solving such systems can be found in the literature, but these are generally not as efficient as solving a symmetric, positive-definite system. We will therefore use a so-called Schur-complement method to reduce the mixed hybrid system to a positive-definite system. The Schur-complement method basically consists of using a block-wise Gaussian

[1] If you are not familiar with the notion of a trace operator, think of it as the values of a function along the boundary of the domain this function is defined on.

elimination of (8.15) to form a positive-definite system (the Schur complement) for the face pressures,

$$\left(\boldsymbol{D}^{\mathsf{T}}\boldsymbol{B}^{-1}\boldsymbol{D} - \boldsymbol{F}^{\mathsf{T}}\boldsymbol{L}^{-1}\boldsymbol{F}\right)\boldsymbol{\pi} = \boldsymbol{F}^{\mathsf{T}}\boldsymbol{L}^{-1}\boldsymbol{q}. \tag{8.16}$$

Here, $\boldsymbol{F} = \boldsymbol{C}^{\mathsf{T}}\boldsymbol{B}^{-1}\boldsymbol{D}$ and $\boldsymbol{L} = \boldsymbol{C}^{\mathsf{T}}\boldsymbol{B}^{-1}\boldsymbol{C}$. Given the face pressures, the cell pressures and fluxes can be reconstructed by back-substitution, i.e., by solving

$$\boldsymbol{L}\boldsymbol{p} = \boldsymbol{q} + \boldsymbol{F}\boldsymbol{\pi}, \qquad \boldsymbol{B}\boldsymbol{u} = \boldsymbol{C}\boldsymbol{p} - \boldsymbol{D}\boldsymbol{\pi}. \tag{8.17}$$

Unlike the mixed method, the hybrid method has a so-called explicit flux representation, which means that the inter-cell fluxes can be expressed as a linear combination of neighboring values for the pressure. This property is particularly useful in fully implicit discretizations of time-dependent problems, as discussed in Chapter 7.

This is all we will discuss about mixed and mixed-hybrid methods herein. Readers interested in learning more about mixed finite-element methods, and how to solve the corresponding linear systems are advised to consult some of the excellent books on the subject, for instance [14, 13, 15]. In the rest of the chapter, we will instead discuss how ideas from mixed-hybrid finite-element methods can be used to formulate finite-volume methods that are consistent and hence convergent on grids that are not K-orthogonal.

8.2 Consistent Methods on Mixed Hybrid Form

In this section, we will present a large class of consistent, finite-volume methods. Our formulation will borrow several ideas from the mixed methods, but will be much simpler to formulate, and implement, for general polygonal and polyhedral grids. For simplicity, we will assume that all methods can be written on the following local form

$$\boldsymbol{v}_i = \boldsymbol{T}_i(\boldsymbol{e}p_i - \boldsymbol{\pi}_i), \tag{8.18}$$

where $\boldsymbol{v}_i$ is a vector of all fluxes associated with a cell Ω_i, $\boldsymbol{e}_i = (1, \ldots, 1)^{\mathsf{T}}$ has one unit value per face of the cell, $\boldsymbol{\pi}_i$ is a vector of face pressures, and $\boldsymbol{T}_i$ is a matrix of one-sided transmissibilities. The local discretization (8.18) can alternatively be written as

$$\boldsymbol{M}_i\boldsymbol{v}_i = \boldsymbol{e}p_i - \boldsymbol{\pi}_i. \tag{8.19}$$

Consistent with the discussion in Section 8.1, the matrix $\boldsymbol{M}$ is referred to as the local inner product. From the local discretizations (8.18) or (8.18) on each cell, we will derive a linear system of discrete global equations written on mixed or hybridized mixed form. Both forms are supported in MRST, but herein we will only discuss the hybrid form for brevity. In the two-point

method, the linear system was developed by combining mass conservation and Darcy's law into one second-order discrete equation for the pressure. In the mixed formulation, the mass conservation and Darcy's law are kept as separate first-order equations that together form a coupled system for pressure and face fluxes. In the hybrid formulation, the continuity of pressures across cell faces is introduced as a third equation that together with mass conservation and Darcy's law constitute a coupled system for pressure, face pressure, and fluxes. Not all consistent methods need to, or can be formulated on this form. However, using the mixed hybrid formulation will enable us to give a uniform presentation of a large class of schemes that also includes the coarse-scale formulation of several multiscale methods, see e.g., [26].

Going back to the TPFA method formulated in Section 5.4.1, it follows immediately from (5.49) that the method can be written as in (8.18) and that the resulting $\boldsymbol{T}_i$ matrix is diagonal with entries

$$(\boldsymbol{T}_i)_{kk} = \vec{n}_{ik} \cdot \mathsf{K}\vec{c}_{ik}/|\vec{c}_{ik}|^2, \tag{8.20}$$

where the length of the normal vector $\vec{n}_{ik}$ is assumed to be equal to the area of the corresponding face. The equivalent form (8.19) follows trivially by setting $\boldsymbol{M}_i = \boldsymbol{T}_i^{-1}$. Examples of consistent methods that can be written in the form (8.18) or (8.19) include the lowest-order Raviart–Thomas methods seen in the previous section, multipoint flux approximation (MPFA) schemes [6, 25, 4], and recently developed mimetic finite-difference methods [16]. For all these methods, the corresponding $\boldsymbol{T}_i$ and $\boldsymbol{M}_i$ will be full matrices. We will come back to more details about specific schemes later in the chapter.

To derive the global linear system on mixed hybrid form, we augment (8.19) with flux and pressure continuity across cell faces. By assembling the contributions from all cells in the grid, we get the following linear system

$$\begin{bmatrix} \boldsymbol{B} & \boldsymbol{C} & \boldsymbol{D} \\ \boldsymbol{C}^{\mathsf{T}} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{D}^{\mathsf{T}} & \boldsymbol{0} & \boldsymbol{0} \end{bmatrix} \begin{bmatrix} \boldsymbol{v} \\ -\boldsymbol{p} \\ \boldsymbol{\pi} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{q} \\ \boldsymbol{0} \end{bmatrix}, \tag{8.21}$$

where the first row in the block-matrix equation corresponds to (8.19) for all grid cells. The vector $\boldsymbol{v}$ contains the outward fluxes associated with half faces ordered cell-wise so that fluxes over interior interfaces in the grid appear twice, once for each half face, with opposite signs. Likewise, the vector $\boldsymbol{p}$ contains the cell pressures, and $\boldsymbol{\pi}$ the face pressures. The matrices $\boldsymbol{B}$ and $\boldsymbol{C}$ are block diagonal with each block corresponding to a cell,

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{M}_1 & 0 & \dots & 0 \\ 0 & \boldsymbol{M}_2 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & \boldsymbol{M}_n \end{bmatrix}, \qquad \boldsymbol{C} = \begin{bmatrix} \boldsymbol{e}_1 & 0 & \dots & 0 \\ 0 & \boldsymbol{e}_2 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & \boldsymbol{e}_n \end{bmatrix}, \tag{8.22}$$

where $\boldsymbol{M}_i = \boldsymbol{T}_i^{-1}$. Similarly, each column of $\boldsymbol{D}$ corresponds to a unique interface in the grid and has two unit entries for interfaces between cells in

the interior of the grid. The two nonzero entries appear at the indexes of the corresponding half-faces in the cell-wise ordering. Similarly, $\boldsymbol{D}$ has a single nonzero entry in each column that corresponds to an interface between a cell and the exterior.

The hybrid system (8.21) is obviously much larger than the linear system obtained for the standard TPFA method, but can be reduced to a positive-definite system for the face pressures, as discussed in Section 8.1.3, and then solved using either MATLAB's standard linear solvers or a highly-efficient, third-party solver such as the agglomeration multigrid solver, AGMG [47, 7]. From the expressions in (8.16) and (8.17), we see that in order to compute the Schur complement to form the reduced linear system for the face pressures, as well as to reconstruct cell pressures and face fluxes, we only need $\boldsymbol{B}^{-1}$ in the solution procedure above. Moreover, the matrix $\boldsymbol{L}$ is by construction diagonal and computing fluxes is therefore an inexpensive operation. Many schemes—including the mimetic method, the MPFA-O method, and the standard TPFA scheme—yield algebraic approximations for the $\boldsymbol{B}^{-1}$ matrix. Thus, (8.21) encompasses a family of discretization schemes whose properties are determined by the choice of $\boldsymbol{B}$, which we will discuss in more detail in Section 8.3.

However, before digging into details about specific, consistent methods, we revisit the example discussed on page 187 to demonstrate how one easily can replace the TPFA method by a consistent method and thereby significantly reduce the grid-orientation effects seen in Figure 8.2. In MRST, the mimetic methods are implemented in a separate module `mimetic` that is not part of the core functionality. In a script that solves a flow problem, one first loads the `mimetic` module by calling

```
mrstModule add mimetic;
```

and then continues to replace the two-point computation of half-face transmissibilities

```
hT   = computeTrans(G, rock);
```

by a call that constructs the local mimetic half-transmissibility $\boldsymbol{T}_i$ or its equivalent inner product $\boldsymbol{M}_i = \boldsymbol{T}_i^{-1}$ for each cell

```
S = computeMimeticIP(G, rock);
```

Likewise, each call to

```
state = incompTPFA(state, G, hT, fluid);
```

which assembles and solves the symmetric, positive-definite, two-point system is replaced by a call to

```
state = solveIncompFlow(state, G, S, fluid);
```

which will assemble and solve the global mixed hybrid system. (The routine can also be set to assemble a mixed system, and in certain cases also

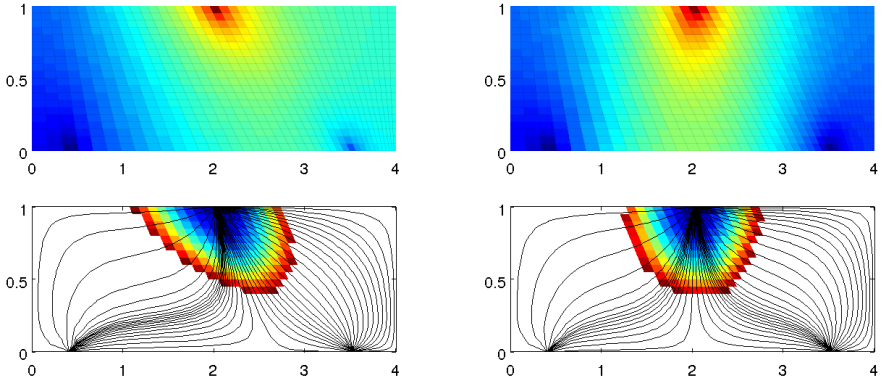


Fig. 8.5. Solution of a symmetric flow problem in a homogeneous domain using the TPFA method (left) and the mimetic method (right) on a skew grid that is not K-orthogonal.

a TPFA-type system.) In Figure 8.5 we have applied the mimetic solver to the exact same setup as in Figure 8.2. The approximate solution computed by the mimetic method is almost symmetric and represents a significant improvement compared with the TPFA method. In particular, the error in the travel time between injector and producers is reduced from 17% for TPFA to less than 2% for the mimetic method. Moreover, repeating a similar grid refinement study as reported in Figure 8.2 verifies that the consistent mimetic method converges towards the correct solution.

The few calls outlined above constitutes all one needs to obtain a consistent discretization on general polygonal and polyhedral grids, and readers who are not interested in getting to know the inner details of various consistent methods can safely jump to the next chapter.

8.3 The Mimetic Method

Mimetic finite-difference methods are examples of so-called *compatible spatial discretizations* that are constructed so that they not only provide accurate approximation of the mathematical models but also inherit or mimic fundamental properties of the differential operators and mathematical solutions they approximate. Examples of properties include conservation, symmetries, vector calculus identities, etc. Such methods have become very popular in recent years and are currently used in wide range of applications, see e.g., [11].

The mimetic methods to be discussed in the following can be seen as a finite-volume generalization of finite-differences or (low-order) mixed-finite element methods to general polyhedral grids. The methods are defined in a way that introduces a certain freedom of construction that naturally leads

to a family of methods. By carefully picking the parameters that are needed to fully specify a method, one can construct mimetic methods that coincide with other known methods, or reduce to these methods (e.g., the two-point method, the RT0 mixed finite-element method, or the MPFA-O multipoint method) on certain types of grids.

The mimetic methods discussed herein, can all be written on the equivalent forms (8.18) or (8.19) and are constructed so that they are exact for linear pressure fields and give a symmetric positive-definite matrix $\boldsymbol{M}$. In addition, the methods use discrete pressures and fluxes associated with cell and face centroids, respectively, and consequently resemble finite-difference methods.

A linear pressure field can be written in the form $p = \vec{x} \cdot \vec{a} + b$ for a constant vector $\vec{a}$ and scalar b, giving a Darcy velocity equal $\vec{v} = -\mathsf{K}\vec{a}$. Let $\vec{n}_{ik}$ denote the area-weighted normal vector to Γ_{ik} and $\vec{c}_{ik}$ be the vector pointing from the centroid of cell Ω_i to the centroid of face Γ_{ik}, as seen in Figure 5.4. Using Darcy's law and the notion of a one-sided transmissibility $\boldsymbol{T}_{ik}$ (which will generally not be the same as the two-point transmissibility), the flux and pressure drop can be related as follows,

$$v_{ik} = -\vec{n}_{ik}\mathsf{K}\vec{a} = T_{ik}(p_i - \pi_{ik}) = -T_{ik}\vec{c}_{ik} \cdot \vec{a}. \tag{8.23}$$

To get this relations in the one of the local forms (8.18) and (8.19) we collect all vectors $\vec{c}_{ik}$ and $\vec{n}_{ik}$ defined for a specific cell Ω_i as rows in two matrices $\boldsymbol{C}_i$ and $\boldsymbol{N}_i$. Because the relation (8.23) is required to hold for all linear pressure drops, i.e., for an arbitrary vector $\vec{a}$, we see that the matrices $\boldsymbol{M}_i$ and $\boldsymbol{T}_i$ must satisfy the following consistency conditions

$$\boldsymbol{MNK} = \boldsymbol{C}, \qquad \boldsymbol{NK} = \boldsymbol{TC}, \tag{8.24}$$

where we for simplicity have dropped the subscript i that identifies the cell number. These general equations give us (quite) large freedom in how to specify a specific discretization method: any method in which the local inner product $\boldsymbol{M}_i$, or equivalently the local transmissibility matrix $\boldsymbol{T}_i$, is positive definite and satisfies (8.24) will be a consistent, first-order accurate discretization. In the rest of the section, we will discuss various choices of valid inner products $\boldsymbol{M}_i$ or reverse inner products $\boldsymbol{T}_i$.

8.3.1 General Family of Inner Products

The original article [16] that first introduced the mimetic methods considered herein, discussed inner products for discrete velocities. However, as we have seen in previous chapters, it is more common in the simulation of flow in porous media to consider inter-cell fluxes as the primary unknowns. We will therefore henceforth consider inner products of fluxes rather than velocities. The relation between the two is trivial: an inner product of velocities becomes an inner product for fluxes by pre- and post-multiplying by the inverse of the area of the corresponding cell faces. In other words, if $\boldsymbol{A}$ is a diagonal matrix

with element A_{jj} equal the are of the j-th face, then the flux inner product $\boldsymbol{M}_{\text{flux}}$ is related to the velocity inner product $\boldsymbol{M}_{\text{vel}}$ through

$$\boldsymbol{M}_{\text{flux}} = \boldsymbol{A}^{-1}\boldsymbol{M}_{\text{vel}}\boldsymbol{A}^{-1}. \tag{8.25}$$

To simplify the derivation of valid solutions, we start by stating the following geometrical property which relates $\boldsymbol{C}$ and $\boldsymbol{N}$ as follows (for a proof, see [16]):

$$\boldsymbol{C}^{\mathsf{T}}\boldsymbol{N} = \boldsymbol{V} = \text{diag}(|\Omega_i|). \tag{8.26}$$

Next, we multiply the left matrix equation in (8.24) by $\boldsymbol{K}^{-1}\boldsymbol{C}^{\mathsf{T}}\boldsymbol{N}$

$$\boldsymbol{C}\left(\boldsymbol{K}^{-1}\boldsymbol{C}^{\mathsf{T}}\boldsymbol{N}\right) = \left(\boldsymbol{M}\boldsymbol{N}\boldsymbol{K}\right)\left(\boldsymbol{K}^{-1}\boldsymbol{C}^{\mathsf{T}}\boldsymbol{N}\right) = \boldsymbol{M}\boldsymbol{N}\boldsymbol{C}^{\mathsf{T}}\boldsymbol{N} = \boldsymbol{M}\boldsymbol{N}\boldsymbol{V},$$

from which it follows that there is a family of valid solution that has the form

$$\boldsymbol{M} = \frac{1}{|\Omega_i|}\boldsymbol{C}\boldsymbol{K}^{-1}\boldsymbol{C}^{\mathsf{T}} + \boldsymbol{M}_2, \tag{8.27}$$

where $\boldsymbol{M}_2$ is a matrix defined such that $\boldsymbol{M}_2\boldsymbol{N} = \boldsymbol{0}$, i.e., any matrix whose rows lie in the left nullspace of $\boldsymbol{N}^{\mathsf{T}}$. Moreover, to make a sensible method we must require that $\boldsymbol{M}$ is symmetric positive definite. In other words, any symmetric and positive definite inner product that fulfills these requirements can be represented in two alternative compact forms,

$$\begin{aligned}\boldsymbol{M} &= \frac{1}{|\Omega_i|}\boldsymbol{C}\boldsymbol{K}^{-1}\boldsymbol{C}^{\mathsf{T}} + \boldsymbol{Q}_N^{\perp}\boldsymbol{S}_M{\boldsymbol{Q}_N^{\perp}}^{\mathsf{T}} \\ &= \frac{1}{|\Omega_i|}\boldsymbol{C}\boldsymbol{K}^{-1}\boldsymbol{C}^{\mathsf{T}} + \boldsymbol{P}_N^{\perp}\boldsymbol{S}_M\boldsymbol{P}_N^{\perp},\end{aligned} \tag{8.28}$$

where $\boldsymbol{S}_M$ denotes any symmetric, positive definite matrix, $\boldsymbol{Q}_N^{\perp}$ is an orthonormal basis for the left null space of $\boldsymbol{N}^{\mathsf{T}}$, and $\boldsymbol{P}_N^{\perp}$ is the null-space projection $\boldsymbol{I} - \boldsymbol{Q}_N{\boldsymbol{Q}_N}^{\mathsf{T}}$ in which $\boldsymbol{Q}_N$ is a basis for the spaces spanned by the columns of $\boldsymbol{N}$. Similarly, we can derive a closed expression for the inverse inner product $\boldsymbol{T}$ by multiplying the right matrix equation in (8.24) by $\boldsymbol{V}^{-1}$ from the left and by $\boldsymbol{V}$ from the right and using the identity (8.26),

$$\boldsymbol{T}\boldsymbol{C} = \boldsymbol{V}^{-1}\left(\boldsymbol{N}\boldsymbol{K}\right)\boldsymbol{V} = \boldsymbol{V}^{-1}\left(\boldsymbol{N}\boldsymbol{K}\right)\left(\boldsymbol{C}^{\mathsf{T}}\boldsymbol{N}\right)^{\mathsf{T}} = \boldsymbol{V}^{-1}\left(\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}}\right)\boldsymbol{C}$$

By the same argument as for $\boldsymbol{M}$, mutatis mutandis, we obtain the following family of inverse inner products,

$$\begin{aligned}\boldsymbol{T} &= \frac{1}{|\Omega_i|}\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}} + \boldsymbol{Q}_C^{\perp}\boldsymbol{S}_T{\boldsymbol{Q}_C^{\perp}}^{\mathsf{T}} \\ &= \frac{1}{|\Omega_i|}\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}} + \boldsymbol{P}_C^{\perp}\boldsymbol{S}_T\boldsymbol{P}_C^{\perp},\end{aligned} \tag{8.29}$$

where $\boldsymbol{Q}_C^{\perp}$ is an orthonormal basis for the left nullspace of $\boldsymbol{C}^{\mathsf{T}}$ and $\boldsymbol{P}_C^{\perp} = \boldsymbol{I} - \boldsymbol{Q}_C{\boldsymbol{Q}_C}^{\mathsf{T}}$ is the corresponding nullspace projection.

The matrices $\boldsymbol{M}$ and $\boldsymbol{T}$ in (8.28) and (8.29) are evidently symmetric, so we only need to prove that they are positive definite. We start by writing the inner product as $\boldsymbol{M} = \boldsymbol{M}_1 + \boldsymbol{M}_2$, and observe that each of these matrices are positive semi-definite and our result follows if we can prove that this implies that $\boldsymbol{M}$ is positive definite. Let $\boldsymbol{z}$ be an arbitrary nonzero vector which we split uniquely as $\boldsymbol{z} = \boldsymbol{N}\boldsymbol{x} + \boldsymbol{y}$, where $\boldsymbol{x}$ lies in the column space of $\boldsymbol{N}$ and $\boldsymbol{y}$ lies in the left nullspace of $\boldsymbol{N}^{\mathsf{T}}$. If $\boldsymbol{y}$ is zero, we have

$$\begin{aligned}\boldsymbol{z}^{\mathsf{T}}\boldsymbol{M}\boldsymbol{z} &= \boldsymbol{x}^{\mathsf{T}}\boldsymbol{N}^{\mathsf{T}}\boldsymbol{M}_1\boldsymbol{N}\boldsymbol{x} \\ &= |\Omega_i|^{-1}\boldsymbol{x}^{\mathsf{T}}\left(\boldsymbol{N}^{\mathsf{T}}\boldsymbol{C}\right)\boldsymbol{K}^{-1}\left(\boldsymbol{C}^{\mathsf{T}}\boldsymbol{N}\right)\boldsymbol{x} = |\Omega_i|\,\boldsymbol{x}^{\mathsf{T}}\boldsymbol{K}^{-1}\boldsymbol{x} > 0\end{aligned}$$

because $\boldsymbol{K}^{-1}$ is a positive definite matrix. If $\boldsymbol{y}$ is nonzero, we have

$$\boldsymbol{z}^{\mathsf{T}}\boldsymbol{M}\boldsymbol{z} = \boldsymbol{z}^{\mathsf{T}}\boldsymbol{M}_1\boldsymbol{z} + \boldsymbol{y}^{\mathsf{T}}\boldsymbol{M}_2\boldsymbol{y} > 0$$

because $\boldsymbol{z}^{\mathsf{T}}\boldsymbol{M}_1\boldsymbol{z} \geq 0$ and $\boldsymbol{y}^{\mathsf{T}}\boldsymbol{M}_2\boldsymbol{y} > 0$. An analogous argument holds for the matrix $\boldsymbol{T}$, and hence we have proved that (8.28) and (8.29) give a well-defined family of consistent discretizations.

So far, we have not put any restrictions on the matrices $\boldsymbol{S}_M$ and $\boldsymbol{S}_T$ that will determine the specific methods, except for requiring that they should be positive definite. In addition, they these matrices should mimic the scaling properties of the continuous equation, which is invariant under affine transformations of space and permeability,

$$\vec{x} \mapsto \boldsymbol{\sigma}\vec{x} \quad \text{and} \quad \mathsf{K} \mapsto \boldsymbol{\sigma}^{\mathsf{T}}\mathsf{K}\boldsymbol{\sigma}, \tag{8.30}$$

To motivate how we should choose the matrices $\boldsymbol{S}_M$ and $\boldsymbol{S}_T$ to mimic these scaling properties, we will look at a simple 1D example:

Example 8.1. Consider the a grid cell $x \in [-1, 1]$, for which $\boldsymbol{N} = \boldsymbol{C} = [1, -1]^{\mathsf{T}}$ and $\boldsymbol{Q}_N^{\perp} = \boldsymbol{Q}_C^{\perp} = \frac{1}{\sqrt{2}}[1, 1]^{\mathsf{T}}$. Hence

$$\begin{aligned}\boldsymbol{M} &= \frac{1}{2}\begin{bmatrix}1\\-1\end{bmatrix}\frac{1}{K}\,[1,-1] + \frac{1}{2}\begin{bmatrix}1\\1\end{bmatrix}S_M\,[1,1], \\ \boldsymbol{T} &= \frac{1}{2}\begin{bmatrix}1\\-1\end{bmatrix}K\,[1,-1] + \frac{1}{2}\begin{bmatrix}1\\1\end{bmatrix}S\,[1,1].\end{aligned} \tag{8.31}$$

The structure of the inner product should be invariant under scaling of K and thus we can write the inner product as a one-parameter family of the form

$$\begin{aligned}\boldsymbol{M} &= \frac{1}{2K}\left(\begin{bmatrix}1&-1\\-1&1\end{bmatrix} + \frac{2}{t}\begin{bmatrix}1&1\\1&1\end{bmatrix}\right), \\ \boldsymbol{T} &= \frac{K}{2}\left(\begin{bmatrix}1&-1\\-1&1\end{bmatrix} + \frac{t}{2}\begin{bmatrix}1&1\\1&1\end{bmatrix}\right).\end{aligned} \tag{8.32}$$

Having established a plausible way of scaling the inner products, we are now in a position to go through various specific choices that are implemented in the `mimetic` module of MRST and look at correspondence between these methods and the standard two-point method, the lowest-order RT0 mixed method, and the MPFA-O method. Our discussion follows [38].

8.3.2 General Parametric Family

Motivated by the example above, we propose to choose the matrix $\boldsymbol{S}_T$ as the diagonal of the first matrix term in (8.29) so that these two terms scale similarly under transformations of the type (8.30). Using this definition of $\boldsymbol{S}_T$ and that $\boldsymbol{M}$ should be equal to $\boldsymbol{T}^{-1}$ suggests the following general family of inner products that only differ in the constant in front of the second (regularization) matrix term:

$$\begin{aligned} \boldsymbol{M} &= \frac{1}{|\Omega_i|}\boldsymbol{C}\boldsymbol{K}^{-1}\boldsymbol{C}^{\mathsf{T}} + \frac{|\Omega_i|}{t}\boldsymbol{P}_N^{\perp}\operatorname{diag}(\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}})^{-1}\boldsymbol{P}_N^{\perp}, \\ \boldsymbol{T} &= \frac{1}{|\Omega_i|}\Big[\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}} + t\,\boldsymbol{P}_C^{\perp}\operatorname{diag}(\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}})\boldsymbol{P}_C^{\perp}\Big]. \end{aligned} \tag{8.33}$$

In MRST, this family of inner products is called 'ip_qfamily' and the parameter t is supplied in a separate option:

```
S = computeMimeticIP(G, rock, 'InnerProduct', 'ip_qfamily', 'qparam', t);
```

As we will see below, mimetic inner products that reduce to the standard TPFA method or the RT0 mixed finite-element method on simple grids are members of this family.

8.3.3 Two-Point Type Methods

A requirement of the two-point method is that the transmissibility matrix $\boldsymbol{T}$ (and hence also the matrix $\boldsymbol{M}$ of the inner product) should be diagonal. Looking at (8.24), we see that this is only possible if the vectors $\mathsf{K}\vec{N}_{ik}$ and $\vec{c}_{ik}$ are parallel, which is the exact same condition for K-orthogonality that we argued was sufficient to guarantee consistency of the method on page 8. An explicit expression for the diagonal entries of the two-point method on K-orthogonal grids has already been given in (8.20). Strictly speaking, this relation does not always yield a positive value for the two-point transmissibility on any grid. For instance, for corner-point grids it is normal to define the face centroids as the arithmetic mean of the associated corner-point nodes and the cell centroids as the arithmetic mean of the centroids top and bottom cell faces, which for most grids will guarantee a positive transmissibility.

The extension of the two-point to non-orthogonal grids is not unique. Here, we will present a mimetic method that coincides with the two-point methods in this method's region of validity, while at the same time giving a valid mimetic inner product and hence a consistent discretization for all types of grids and permeability tensors. One advantage of this method, compared with multipoint flux-approximation methods, is that the implementation is simpler for general unstructured grids.

The most instructive way to introduce the general method is to look at a simple example that will motivate the general construction.

Example 8.2. We consider the grid cell $[-1,1] \times [-1,1]$ in 2D and calculate the components that make up the inverse inner product $\boldsymbol{T}$ for a diagonal and a full permeability tensor K and compare with the corresponding two-point discretization. We start by setting up the permeability and the geometric properties of the cell

```
K1 = eye(2);  K2 = [1 .5; .5 1];
C = [-1 0; 1 0; 0 -1; 0 1]; N = 2*C; vol = 4;
```

Using the definition (8.20), we see that the transmissibility matrix corresponding resulting from the standard two-point discretization can be computed as:

```
T = diag(diag(N*K*C')./sum(C.*C,2))
```

The result is `T=diag([2 2 2 2])` for both permeability tensors, which clearly demonstrates the lack of consistency for this scheme. To construct the inverse inner product, we start by computing the nullspace projection

```
Q = orth(C);
P = eye(size(C,1)) - Q*Q';
```

We saw above that as a simple means of providing a positive definite matrix $\boldsymbol{S}_T$ that scales similarly to the first term in the definition of $\boldsymbol{T}$ in (8.29), we could choose $\boldsymbol{S}_T$ as a multiple of the diagonal of this matrix:

```
W = (N * K * N') ./ vol;
St = diag(diag(W));
```

Collecting our terms, we see that the inverse inner product will be made up of the following two terms for the case with the diagonal permeability tensor:

```
W =  1   -1    0    0      P*St*P =  0.5   0.5    0    0
    -1    1    0    0                0.5   0.5    0    0
     0    0    1   -1                  0     0   0.5  0.5
     0    0   -1    1                  0     0   0.5  0.5
```

If we now define the inverse inner product as

```
T = W + 2*P*St*P
```

the resulting method will coincide with the diagonal transmissibility matrix for diagonal permeability and give a full matrix for the tensor permeability,

```
T1 = 2.0    0    0    0     T2 =  2.0    0   0.5  -0.5
       0  2.0    0    0           0    2.0  -0.5   0.5
       0    0  2.0    0         0.5  -0.5   2.0     0
       0    0    0  2.0        -0.5   0.5     0   2.0
```

Altogether, we have derived a discrete inner product that generalizes the two-point method to full tensor permeabilities on 2D Cartesian grids and gives a consistent discretization also when the grid is not K-orthogonal.

Motivated by the above example, we define a quasi-two-point inner product for general grids that simplifies to the standard TPFA method on Cartesian grids with diagonal permeability tensor:

$$\begin{aligned} \boldsymbol{M} &= \frac{1}{|\Omega_i|}\boldsymbol{C}\boldsymbol{K}^{-1}\boldsymbol{C}^{\mathsf{T}} + \frac{|\Omega_i|}{2}\boldsymbol{P}_N^{\perp}\operatorname{diag}(\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}})^{-1}\boldsymbol{P}_N^{\perp}, \\ \boldsymbol{T} &= \frac{1}{|\Omega_i|}\Big[\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}} + 2\,\boldsymbol{P}_C^{\perp}\operatorname{diag}(\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}})\boldsymbol{P}_C^{\perp}\Big]. \end{aligned} \tag{8.34}$$

The observant reader will notice that this is a special case for $t = 2$ of the general family (8.35) of inner products.

In MRST, this inner product is called 'ip_quasitpf' and is obtained by constructing the inner product using the following call sequence

```
S = computeMimeticIP(G, rock, 'InnerProduct', 'ip_quasitpf');
```

For completeness, the `mimetic` module also supplies a standard, diagonal two-point inner product that will not generally be consistent. This is invoked by the following call:

```
S = computeMimeticIP(G, rock, 'InnerProduct', 'ip_tpf');
```

8.3.4 Raviart–Thomas Type Inner Product

To compute the mixed finite-element inner product on cells that are not simplexes or hexahedrons aligned with the coordinate axes, the standard approach is to map the cell back to a unit reference cell on which the basis mixed basis functions are defined and perform the integration there. For a general hexahedral cell in 3D the integral based on such a transformation would conceptually look something like

$$\iiint_{\Omega_i} f(\vec{x})\,d\vec{x} = \iiint_{[0,1]^3} f(\vec{x}(\xi,\eta,\zeta))\,|J(\xi,\eta,\zeta)|\,d\xi d\eta d\zeta$$

where $J = \partial(x,y,z)/\partial(\xi,\eta,\zeta)$ denotes the Jacobian matrix of the coordinate transformation. The determinant of J will generally be nonlinear and it is therefore not possible to develop a mimetic inner product that is equal to the lowest-order, Raviart–Thomas (RT0) inner product on general polygonal or polyhedral cells, and at the same time simpler to compute.

Instead, we will develop an inner product that is equivalent to RT0 on grids that are orthogonal and aligned with the principal axes of the permeability tensor. To motivate the definition of the resulting inner product, we first look at a simple example.

Example 8.3. Let us look at the RT0 basis functions defined on the reference element in two spatial dimensions

$$\vec{\psi}_1 = \begin{pmatrix} 1-x \\ 0 \end{pmatrix}, \quad \vec{\psi}_2 = \begin{pmatrix} x \\ 0 \end{pmatrix}, \quad \vec{\psi}_3 = \begin{pmatrix} 0 \\ 1-y \end{pmatrix}, \quad \vec{\psi}_4 = \begin{pmatrix} 0 \\ y \end{pmatrix}$$

and compute the elements of the corresponding inner product matrix for a diagonal permeability with unit entries, $\boldsymbol{K} = \boldsymbol{I}$. This entails careful treatment of a sticky point: whereas the mixed inner product involves velocities, the mimetic inner product involves fluxes that are are considered positive when going out of a cell and negative when going into the cell. To get comparative results, we will therefore reverse the sign of $\vec{\psi}_1$ and $\vec{\psi}_3$. Using symmetry and the fact that $\vec{\psi}_i \cdot \vec{\psi}_k = 0$ if $i = 1, 2$ and $k = 3, 4$, we only have to compute two integrals:

$$\int_0^1 \int_0^1 \vec{\psi}_1 \cdot \vec{\psi}_1 \, dxdy = \int_0^1 (x-1)^2 \, dx = \tfrac{1}{3}$$
$$\int_0^1 \int_0^1 \vec{\psi}_1 \cdot \vec{\psi}_2 \, dxdy = \int_0^1 (x-1)x \, dx = -\tfrac{1}{6}$$

This means that the RT0 inner product in 2D reads

$$\boldsymbol{M} = \begin{bmatrix} \frac{1}{3} & -\frac{1}{6} & 0 & 0 \\ -\frac{1}{6} & \frac{1}{3} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & -\frac{1}{6} \\ 0 & 0 & -\frac{1}{6} & \frac{1}{3} \end{bmatrix}$$

Similarly as in Example 8.2, we can compute the two matrices that will be used to form the mimetic inner product

```
C  = .5*[-1 0; 1 0; 0 -1; 0 1]; N  = 2*C; vol = 1; K  = eye(2)
Q  = orth(N);
P  = eye(size(C,1)) - Q*Q';
Sm = diag(1 ./ diag(N*K*N'))*vol;
M1 = C*(K\C')./vol
M2 = P*Sm*P
```

The result of this computation is

```
M1 = 0.25  -0.25     0     0      M2 = 0.50   0.50     0     0
    -0.25   0.25     0     0           0.50   0.50     0     0
        0      0  0.25 -0.25              0      0  0.50  0.50
        0      0 -0.25  0.25              0      0  0.50  0.50
```

from which it follows that $\boldsymbol{M}_2$ should be scaled by $\frac{1}{6}$ if the mimetic inner product is to coincide with the RTO mixed inner product.

For a general grid, we define a quasi-RT0 inner product that simplifies to the standard RT0 inner product on orthogonal grids with diagonal permeability tensors, as well as on all other cases that can be transformed to such a grid by an affine transformation of the form (8.30). The quasi-RT0 inner product reads

$$\begin{aligned} \boldsymbol{M} &= \frac{1}{|\Omega_i|} \boldsymbol{C}\boldsymbol{K}^{-1}\boldsymbol{C}^{\mathsf{T}} + \frac{|\Omega_i|}{6} \boldsymbol{P}_N^{\perp} \operatorname{diag}(\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}})^{-1} \boldsymbol{P}_N^{\perp}, \\ \boldsymbol{T} &= \frac{1}{|\Omega_i|} \Big[\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}} + 6\, \boldsymbol{P}_C^{\perp} \operatorname{diag}(\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}}) \boldsymbol{P}_C^{\perp} \Big]. \end{aligned} \tag{8.35}$$

and is a special case of the general family (8.35) of inner products for $t = 6$.

In MRST, this inner product is called 'ip_quasirt' and is obtained by constructing the inner product as follows

```
S = computeMimeticIP(G, rock, 'InnerProduct', 'ip_quasitpf');
```

For completeness, the `mimetic` module also supplies a standard RTO inner product, 'ip_rt' that is only valid on Cartesian grids.

8.3.5 Default Inner Product in MRST

For historic reasons, the default discretization in the `mimetic` module of MRST corresponds to a mimetic method with half-transmissibilities defined by the following expression,

$$\boldsymbol{T} = \frac{1}{|\Omega_i|}\Big[\boldsymbol{N}\boldsymbol{K}\boldsymbol{N}^{\mathsf{T}} + \frac{6}{d}\operatorname{tr}(\boldsymbol{K})\boldsymbol{A}\left(\boldsymbol{I} - \boldsymbol{Q}\boldsymbol{Q}^{\mathsf{T}}\right)\boldsymbol{A}\Big]. \tag{8.36}$$

where $\boldsymbol{A}$ is the diagonal matrix containing face areas and $\boldsymbol{Q}$ is an orthogonal basis for the range of $\boldsymbol{AC}$. The inner product, referred to as 'ip_simple', was inspired by [16] and introduced in [3] to resemble the Raviart–Thomas inner product (they are equal for scalar permeability on orthogonal grids, which can be verified by inspection). Because the inner product is based on velocities, it involves pre- and post-multiplication of (inverse) face areas and is in this sense different from the general class of inner products discussed above. However, it is possible to show that the eigenspace corresponding to the nonzero eigenvalues of the second matrix term is equal to the nullspace for $\boldsymbol{C}$.

8.3.6 Local-Flux Mimetic Method

Another large class of consistent discretization methods that have received a lot of attention in recent years is the multipoint flux-approximation (MPFA) schemes [6, 25, 4]. Discussing different variants of this method is beyond the scope of the current presentation, mainly because efficient implementation of a general class of MPFA schemes requires some additional mappings that are not yet part of the standard grid structure outlined in Section 3.4. However, at the time of writing, work is in progress to expand the grid structure to support the implementation of other MPFA type schemes

Having said this, there *is* a module `mpfa` in MRST that gives a simple implementation of the MPFA-O method. In this implementation, we have utilized the fact that some variants of the method can be formulated as a mimetic method. This was first done by [36, 39] and is called the local-flux mimetic formulation of the MPFA method. In this approach, each face in the grid is subdivided into a set of subfaces, one subface per node that makes up the face. The inner product of the local-flux mimetic method gives exact result for linear flow and is block diagonal with respect to the faces corresponding

to each node of the cell, but it is not symmetric. The block-diagonal property makes it possible to reduce the system into a cell-centered discretisation for the cell pressures. This naturally leads to a method for calculating the MPFA transmissibilities. The crucial point is to have the corner geometry in the grid structure and handle the problems with corners which do not have three unique half faces associated.

The local-flux mimetic implementation of the MPFA-O method can be used with a calling sequence that is similar to the TPFA and the mimetic methods:

```
mrstModule add mpfa;
hT = computeMultiPointTrans(G, rock);
state = incompMPFA(state, G, hT, fluid)
```

However, the implementation is based on a combination of MATLAB and C, which is used to invert small systems, and may not work out of the box on all computers.

References

[1] J. E. Aarnes, V. Kippe, and K.-A. Lie. Mixed multiscale finite elements and streamline methods for reservoir simulation of large geomodels. *Adv. Water Resour.*, 28(3):257–271, 2005. doi:10.1016/j.advwatres.2004.10.007.

[2] J. E. Aarnes, T. Gimse, and K.-A. Lie. An introduction to the numerics of flow in porous media using Matlab. In G. Hasle, K.-A. Lie, and E. Quak, editors, *Geometrical Modeling, Numerical Simulation and Optimisation: Industrial Mathematics at SINTEF*, pages 265–306. Springer Verlag, Berling Heidelberg New York, 2007.

[3] J. E. Aarnes, S. Krogstad, and K.-A. Lie. Multiscale mixed/mimetic methods on corner-point grids. *Comput. Geosci.*, 12(3):297–315, 2008. ISSN 1420-0597. doi:10.1007/s10596-007-9072-8.

[4] I. Aavatsmark. An introduction to multipoint flux approximations for quadrilateral grids. *Comput. Geosci.*, 6:405–432, 2002. doi:10.1023/A:1021291114475.

[5] I. Aavatsmark and R. Klausen. Well index in reservoir simulation for slanted and slightly curved wells in 3d grids. *SPE J.*, 8(01):41–48, 2003.

[6] I. Aavatsmark, T. Barkve, Ø. Bøe, and T. Mannseth. Discretization on non-orthogonal, curvilinear grids for multi-phase flow. *Proc. of the 4th European Conf. on the Mathematics of Oil Recovery*, 1994.

[7] AGMG. Iterative solution with AGgregation-based algebraic MultiGrid, 2012. `http://homepages.ulb.ac.be/~ynotay/AGMG/`.

[8] I. Akervoll and P. Bergmo. A study of Johansen formation located offshore Mongstad as a candidate for permanent CO2 storage. In *European Conference on CCS Research, Development and Demonstration. 10–11 February 2009, Oslo, Norway*, 2009.

[9] J. Alvestad, K. Holing, K. Christoffersen, O. Stava, et al. Interactive modelling of multiphase inflow performance of horizontal and highly deviated wells. In *European Petroleum Computer Conference*. Society of Petroleum Engineers, 1994.

[10] J. Bear. *Dynamics of Fluids in Porous Media*. Dover, 1988. ISBN 0-486-45355-3.

[11] L. Beirao da Veiga, K. Lipnikov, and G. Manzini. *The Mimetic Finite Difference Method for Elliptic Problems*, volume 11 of *MS&A – Modeling, Simulation and Applications*. Springer, 2014. doi:10.1007/978-3-319-02663-3.

[12] C. H. Bischof, H. M. Bücker, B. Lang, A. Rasch, and A. Vehreschild. Combining source transformation and operator overloading techniques to compute derivatives for MATLAB programs. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002)*, pages 65–72, Los Alamitos, CA, USA, 2002. IEEE Computer Society. doi:10.1109/SCAM.2002.1134106.

[13] D. Braess. *Finite elements: Theory fast solvers and applications in solid mechanics*. Cambridge University Press, Cambridge, 1997.

[14] S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods*, volume 15 of *Texts in Applied Mathematics*. Springer–Verlag, New York, 1994.

[15] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods*, volume 15 of *Springer Series in Computational Mathematics*. Springer-Verlag, New York, 1991. ISBN 0-387-97582-9.

[16] F. Brezzi, K. Lipnikov, and V. Simoncini. A family of mimetic finite difference methods on polygonial and polyhedral meshes. *Math. Models Methods Appl. Sci.*, 15:1533–1553, 2005. doi:10.1142/S0218202505000832.

[17] Cayuga Research. Admat. URL `http://www.cayugaresearch.com/admat.html`. [Online; accessed 15-04-2014].

[18] Z. Chen, G. Huan, and Y. Ma. *Computational methods for multiphase flows in porous media*, volume 2 of *Computational Science and Engineering*. Society of Industrial and Applied Mathematics (SIAM), 2006. doi:10.1137/1.9780898718942.

[19] M. A. Christie and M. J. Blunt. Tenth SPE comparative solution project: A comparison of upscaling techniques. *SPE Reservoir Eval. Eng.*, 4:308–317, 2001. doi:10.2118/72469-PA. Url: `http://www.spe.org/csp/`.

[20] C. Cordes and W. Kinzelbach. Continous groundwater velocity fields and path lines in linear, bilinear, and trilinear finite elements. *Water Resour. Res.*, 28(11):2903–2911, 1992.

[21] H. P. G. Darcy. *Les Fontaines Publiques de la Ville de Dijon*. Dalmont, Paris, 1856.

[22] A. Datta-Gupta and M. King. A semianalytic approach to tracer flow modeling in heterogeneous permeable media. *Adv. Water Resour.*, 18: 9–24, 1995.

[23] A. Datta-Gupta and M. J. King. *Streamline Simulation: Theory and Practice*, volume 11 of *SPE Textbook Series*. Society of Petroleum Engineers, 2007.

[24] C. V. Deutsch and A. G. Journel. *GSLIB: Geostatistical software library and user's guide*. Oxford University Press, New York, 2nd edition, 1998.

[25] M. G. Edwards and C. F. Rogers. A flux continuous scheme for the full tensor pressure equation. *Proc. of the 4th European Conf. on the Mathematics of Oil Recovery*, 1994.

[26] Y. Efendiev and T. Y. Hou. *Multiscale Finite Element Methods*, volume 4 of *Surveys and Tutorials in the Applied Mathematical Sciences*. Springer Verlag, New York, 2009.

[27] G. Eigestad, H. Dahle, B. Hellevang, W. Johansen, K.-A. Lie, F. Riis, and E. Øian. Geological and fluid data for modelling CO2 injection in the Johansen formation, 2008. URL http://www.sintef.no/Projectweb/MatMorA/Downloads/Johansen.

[28] G. Eigestad, H. Dahle, B. Hellevang, F. Riis, W. Johansen, and E. Øian. Geological modeling and simulation of CO2 injection in the Johansen formation. *Comput. Geosci.*, 13(4):435–450, 2009. doi:10.1007/s10596-009-9153-y.

[29] R. E. Ewing, R. D. Lazarov, S. L. Lyons, D. V. Papavassiliou, J. Pasciak, and G. Qin. Numerical well model for non-Darcy flow through isotropic porous media. *Comput. Geosci.*, 3(3-4):185–204, 1999. doi:10.1023/A:1011543412675.

[30] M. Fink. Automatic differentiation for Matlab. MATLAB Central, 2007. URL http://www.mathworks.com/matlabcentral/fileexchange/15235-automatic-differentiation-for-matlab. [Online; accessed 15-04-2014].

[31] S. A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Trans. Math. Software*, 32 (2):195–222, 2006.

[32] H. Hægland, H. K. Dahle, K.-A. Lie, and G. Eigestad. Adaptive streamline tracing for streamline simulation on irregular grids. In P. Binning, P. Engesgaard, H. Dahle, G. Pinder, and W. Gray, editors, *Proceedings of the XVI International Conference on Computational Methods in Water Resources*, Copenhagen, Denmark, 18–22 June 2006. URL http://proceedings.cmwr-xvi.org/.

[33] H. Holden and N. Risebro. *Front Tracking for Hyperbolic Conservation Laws*, volume 152 of *Applied Mathematical Sciences*. Springer, New York, 2002.

[34] E. Jimenez, K. Sabir, A. Datta-Gupta, and M. King. Spatial error and convergence in streamline simulation. *SPE J.*, 10(3):221–232, June 2007.

[35] M. King and A. Datta-Gupta. Streamline simulation: A current perspective. *In Situ*, 22(1):91–140, 1998.

[36] R. A. Klausen and A. F. Stephansen. Mimetic MPFA. In *Proc. 11th European Conference on the Mathematics of Oil Recovery, 8-11 Sept., Bergen, Norway*, number A12. EAGE, 2008.

[37] R. A. Klausen, A. F. Rasmussen, and A. Stephansen. Velocity interpolation and streamline tracing on irregular geometries. *Computational Geosciences*, 16:261–276, 2012. doi:10.1007/s10596-011-9256-0.

[38] K.-A. Lie, S. Krogstad, I. S. Ligaarden, J. R. Natvig, H. Nilsen, and B. Skaflestad. Open-source MATLAB implementation of consistent discretisations on complex grids. *Comput. Geosci.*, 16:297–322, 2012. doi:10.1007/s10596-011-9244-4.

[39] K. Lipnikov, M. Shashkov, and I. Yotov. Local flux mimetic finite difference methods. *Numer. Math.*, 112(1):115–152, 2009. doi:10.1007/s00211-008-0203-5.

[40] T. Manzocchi et al. Sensitivity of the impact of geological uncertainty on production from faulted and unfaulted shallow-marine oil reservoirs: objectives and methods. *Petrol. Geosci.*, 14(1):3–15, 2008.

[41] S. Matringe and M. Gerritsen. On accurate tracing of streamlines. In *SPE Annual Technical Conference and Exhibition*, Houston, Texas, USA, 26-29 September 2004. SPE 89920.

[42] S. Matringe, R. Juanes, and H. Tchelepi. Streamline tracing on general triangular or quadrilateral grids. *SPE J.*, 12(2):217–233, June 2007.

[43] W. McIlhagga. Automatic differentiation with Matlab objects. MATLAB Central, mar 2010. URL `http://www.mathworks.com/matlabcentral/fileexchange/26807-automatic-differentiation-with-matlab-objects`. [Online; accessed 15-04-2014].

[44] J. R. Natvig and K.-A. Lie. Fast computation of multiphase flow in porous media by implicit discontinuous Galerkin schemes with optimal ordering of elements. *J. Comput. Phys.*, 227(24):10108–10124, 2008. doi:10.1016/j.jcp.2008.08.024.

[45] J. R. Natvig, K.-A. Lie, B. Eikemo, and I. Berre. An efficient discontinuous Galerkin method for advective transport in porous media. *Adv. Water Resour.*, 30(12):2424–2438, 2007. doi:10.1016/j.advwatres.2007.05.015.

[46] R. Neidinger. Introduction to automatic differentiation and MATLAB object-oriented programming. *SIAM Review*, 52(3):545–563, 2010. doi:10.1137/080743627.

[47] Y. Notay. An aggregation-based algebraic multigrid method. *Electron. Trans. Numer. Anal.*, 37:123–140, 2010.

[48] P.-E. Øren, S. Bakke, and O. J. Arntzen. Extending predictive capabilities to network models. *SPE J.*, 3(4):324–336, 1998.

[49] D. W. Peaceman. Interpretation of well-block pressures in numerical reservoir simulation with nonsquare grid blocks and anisotropic permeability. *Soc. Petrol. Eng. J.*, 23(3):531–543, 1983. doi:10.2118/10528-PA. SPE 10528-PA.

[50] D. W. Peaceman. *Fundamentals of Numerical Reservoir Simulation*. Elsevier Science Inc., New York, NY, USA, 1991. ISBN 0444415785.

[51] D. W. Peaceman et al. Interpretation of well-block pressures in numerical reservoir simulation. *Soc. Petrol. Eng. J.*, 18(3):183—194, 1978.

[52] D. Pollock. Semi-analytical computation of path lines for finite-difference models. *Ground Water*, 26(6):743–750, 1988.

[53] D. K. Ponting. Corner point geometry in reservoir simulation. In P. King, editor, *Proceedings of the 1st European Conference on Mathematics of Oil Recovery, Cambridge, 1989*, pages 45–65, Oxford, July 25–27 1989. Clarendon Press.

[54] M. Prevost, M. Edwards, and M. Blunt. Streamline tracing on curvilinear structured and unstructured grids. *SPE J.*, 7(2):139–148, June 2002.

[55] P. A. Raviart and J. M. Thomas. A mixed finite element method for second order elliptic equations. In I. Galligani and E. Magenes, editors, *Mathematical Aspects of Finite Element Methods*, pages 292–315. Springer–Verlag, Berlin – Heidelberg – New York, 1977.

[56] L. F. Shampine, R. Ketzscher, and S. A. Forth. Using AD to solve BVPs in MATLAB. *ACM Trans. Math. Software*, 31(1):79–94, 2005.

[57] Technische Universität Darmstadt. Automatic Differentiation for Matlab (ADiMat). URL http://www.adimat.de/. [Online; accessed 15-04-2014].

[58] Tomlab Optimization Inc. Matlab Automatic Differentiation (MAD). URL http://matlabad.com/. [Online; accessed 15-04-2014].

[59] A. Verma. ADMAT: Automatic differentiation in MATLAB using object oriented methods. In *SIAM Interdiscplinary Workshop on Object Oriented Methods for Interoperability*, pages 174–183, 1999.