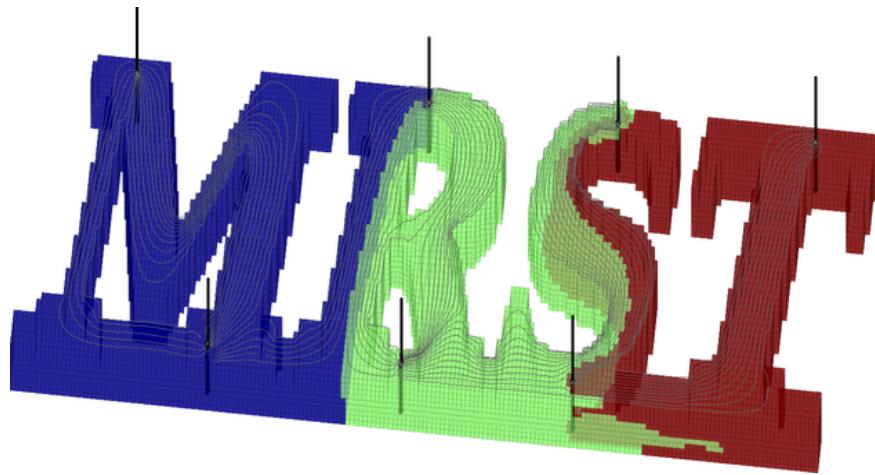


Knut-Andreas Lie

An Introduction to Reservoir Simulation Using MATLAB

User Guide for the Matlab Reservoir Simulation
Toolbox (MRST)

October 19, 2017



SINTEF ICT, Departement of Applied Mathematics
Oslo, Norway

Page: 2

job: nrst-book

macro: svmono.cls

date/time: 19-Oct-2017/1:29

Contents

1	Introduction	5
1.1	Petroleum production	7
1.2	Reservoir simulation	11
1.3	Outline of the book	13
1.4	The first encounter with MRST	18
<hr/>		
Part I Geological Models and Grids		
2	Modelling Reservoir Rocks	23
2.1	Formation of sedimentary rocks	23
2.2	Creation of crude oil and natural gas	28
2.3	Multiscale modelling of permeable rocks	31
2.3.1	Macroscopic models	32
2.3.2	Representative elementary volumes	34
2.3.3	Microscopic models: The pore scale	35
2.3.4	Mesoscopic models	37
2.4	Modelling rock properties	38
2.4.1	Porosity	39
2.4.2	Permeability	40
2.4.3	Other parameters	42
2.5	Property modelling in MRST	43
2.5.1	Homogeneous models	44
2.5.2	Random and lognormal models	44
2.5.3	10 th SPE Comparative Solution Project: Model 2	46
2.5.4	The Johansen Formation	49
2.5.5	SAIGUP: shallow-marine reservoirs	50
3	Grids in Subsurface Modeling	61
3.1	Structured grids	63
3.2	Unstructured grids	69

3.2.1	Delaunay tessellation	69
3.2.2	Voronoi diagrams	72
3.2.3	Other types of tessellations	75
3.2.4	Using an external mesh generator	77
3.3	Stratigraphic grids	80
3.3.1	Corner-point grids	80
3.3.2	2.5D unstructured grids	91
3.4	Grid structure in MRST	93
3.5	Examples of more complex grids	103
3.5.1	Control-point and boundary-aligned grids	110

Part II Single-Phase Flow

4	Mathematical Models and Basic Discretizations	119
4.1	Fundamental concept: Darcy's law	119
4.2	General flow equations for single-phase flow	121
4.3	Auxiliary conditions and equations	126
4.3.1	Boundary and initial conditions	126
4.3.2	Injection and production wells	127
4.3.3	Field lines and time-of-flight	133
4.3.4	Tracers and volume partitions	134
4.4	Basic finite-volume discretizations	136
4.4.1	Two-point flux-approximation	136
4.4.2	Discrete div and grad operators	140
4.4.3	Time-of-flight and tracer	145
5	Incompressible Solvers	147
5.1	Basic data structures in a simulation model	148
5.1.1	Fluid properties	148
5.1.2	Reservoir states	149
5.1.3	Fluid sources	149
5.1.4	Boundary conditions	150
5.1.5	Wells	152
5.2	Incompressible two-point pressure solver	154
5.3	Upwind solver for time-of-flight and tracer	157
5.4	Simulation examples	160
5.4.1	Quarter five-spot	161
5.4.2	Boundary conditions	165
5.4.3	Structured versus unstructured stencils	169
5.4.4	Using Peaceman well models	174

6	Consistent Discretizations on Polyhedral Grids	179
6.1	The mixed finite-element method	182
6.1.1	Continuous formulation	182
6.1.2	Discrete formulation	184
6.1.3	Hybrid formulation	187
6.2	Consistent methods on mixed hybrid form	189
6.3	The mimetic method	193
6.3.1	General family of inner products	194
6.3.2	General parametric family	196
6.3.3	Two-point type methods	196
6.3.4	Raviart–Thomas type inner product	198
6.3.5	Default inner product in MRST	200
6.3.6	Local-flux mimetic method	201
6.3.7	Monotonicity	202
7	Single-Phase Flow and Rapid Prototyping	205
7.1	Implicit discretization	205
7.2	A simulator based on automatic differentiation	207
7.2.1	Model setup and initial state	207
7.2.2	Discrete operators and equations	209
7.2.3	Well model	211
7.2.4	The simulation loop	212
7.3	Pressure-dependent viscosity	216
7.4	Non-Newtonian fluid	218
7.5	Thermal effects	224

Part III Multiphase Flow

8	Mathematical Models for Multiphase Flow	235
8.1	New physical properties and phenomena	236
8.1.1	Saturation	237
8.1.2	Wettability	238
8.1.3	Capillary pressure	239
8.1.4	Relative permeability	244
8.2	Flow equations for multiphase flow	248
8.2.1	Single-component phases	248
8.2.2	Multicomponent phases	250
8.2.3	Black-oil models	250
8.3	Model reformulations for immiscible two-phase flow	252
8.3.1	Pressure formulation	253
8.3.2	Fractional flow formulation in phase pressure	253
8.3.3	Fractional flow formulation in global pressure	259
8.3.4	Fractional flow formulation in phase potential	260
8.3.5	Richards' equation	261

VIII Contents

8.4	The Buckley–Leverett theory of 1D displacements	263
8.4.1	Horizontal displacement	263
8.4.2	Gravity segregation	269
8.4.3	Front tracking: semi-analytical solutions	272
9	Discretizing Hyperbolic Transport Equations	279
9.1	A new solution concept: entropy-weak solutions	280
9.2	Conservative finite-volume methods	282
9.3	A few classical schemes	283
9.3.1	Centered schemes	283
9.3.2	Upwind or Godunov schemes	286
9.3.3	Comparison of centered and upwind schemes	287
9.3.4	Implicit schemes	291
9.4	Convergence of conservative methods	295
9.5	High-resolution schemes	297
9.5.1	Flux-limiter schemes	297
9.5.2	Slope-limiter schemes	298
9.5.3	Semi-discrete schemes	303
9.6	Discretization on unstructured polyhedral grids	306
10	Solvers for Incompressible Immiscible Flow	309
10.1	Fluid objects for multiphase flow	310
10.2	Sequential solution procedures	312
10.2.1	Pressure solvers	313
10.2.2	Saturation solvers	314
10.3	Simulation examples	318
10.3.1	Buckley–Leverett displacement	319
10.3.2	Inverted gravity column	321
10.3.3	Homogeneous quarter five-spot	324
10.3.4	Heterogeneous quarter five-spot: viscous fingering	328
10.3.5	Buoyant migration of CO ₂ in a sloping sandbox	331
10.3.6	Water coning and gravity override	334
10.3.7	The effect of capillary forces – capillary fringe	340
10.3.8	Norne: simplified simulation of a real-field model	344
10.4	Numerical errors	347
10.4.1	Splitting errors	347
10.4.2	Grid-orientation errors	352
11	Compressible Multiphase Flow	359
11.1	Two-phase flow without phase transfer	365
11.2	Three-phase relative permeabilities	371
11.3	PVT behavior of petroleum fluids	383
11.3.1	Phase diagrams	383
11.3.2	Reservoir types and their phase behavior during recovery	386
11.3.3	PVT and fluid properties in black-oil models	394

11.4	Phase behavior in ECLIPSE input decks	403
11.5	The black-oil equations	411
11.5.1	The aqueous component	412
11.5.2	The oil component	414
11.5.3	The gas component	416
11.5.4	Appearence and disappearance of phases	416
11.6	Well models	418
11.6.1	Inflow performance relationships	418
11.6.2	Multisegment wells	420
11.7	Black-oil simulation with MRST	423
11.7.1	Simulating the SPE 1 benchmark case	424
11.7.2	Comparison against a commercial simulator	430
11.7.3	Limitations and potential pitfalls	430
12	The AD-OO framework	437
12.1	Overview of the simulator framework	438
12.2	Model hierarchy	444
12.2.1	PhysicalModel – generic physical models	446
12.2.2	ReservoirModel – basic reservoir models	450
12.2.3	Black-oil models	455
12.2.4	Models of wells and production facilities	457
12.3	Solving the discrete model equations	459
12.3.1	Assembly of linearized systems	459
12.3.2	Nonlinear solvers	461
12.3.3	Selection of time-steps	463
12.3.4	Linear solvers	465
12.4	Simulation examples	471
12.4.1	Depletion of a closed/open compartment	471
12.4.2	An undersaturated sector model	472
12.4.3	SPE 1 with a multisegment well	477
12.4.4	The SPE 9 benchmark case	482
12.5	Improving convergence and reducing runtime	492

Part IV Reservoir Engineering Workflows

13	Flow Diagnostics	497
13.1	Flow patterns and volumetric connections	498
13.1.1	Volumetric partitions	500
13.1.2	Time-of-flight per partition region: improved accuracy	502
13.1.3	Well-allocation factors	502
13.2	Measures of dynamic heterogeneity	503
13.2.1	Flow and storage capacity	504
13.2.2	Lorenz coefficient and sweep efficiency	506
13.2.3	Summary of diagnostic curves and measures	508

13.3	Case studies	510
13.3.1	Tarbert formation: volumetric connections	510
13.3.2	Layers of SPE 10: heterogeneity and sweep improvement	514
13.4	Interactive flow diagnostics tools	519
13.4.1	Simple 2D example	522
13.4.2	SAIGUP: flow patterns and volumetric connections	526
14	Grid Coarsening	529
14.1	Partition vectors	530
14.1.1	Uniform partitions	531
14.1.2	Connected partitions	531
14.1.3	Composite partitions	532
14.2	Coarse grid representation in MRST	535
14.2.1	Subdivision of coarse faces	536
14.3	Coarsening of realistic reservoir models	539
14.3.1	The Johansen aquifer	539
14.3.2	The SAIGUP model	542
14.4	General advice and simple guidelines	546
15	Upscaling Petrophysical Properties	549
15.1	Upscaling for reservoir simulation	551
15.2	Upscaling additive properties	553
15.3	Upscaling absolute permeability	555
15.3.1	Averaging methods	556
15.3.2	Flow-based upscaling	562
15.4	Upscaling transmissibility	567
15.5	Global and local-global upscaling	570
15.6	Upscaling examples	573
15.6.1	Flow diagnostics quality measure	573
15.6.2	Model with two rock types	574
15.6.3	SPE 10 with six wells	577
15.6.4	General advice and simple guidelines	581
A	The MATLAB Reservoir Simulation Toolbox	583
A.1	Getting started with the software	584
A.1.1	Core functionality and add-on modules	584
A.1.2	Downloading and installing	587
A.1.3	Exploring the functionality and getting help	588
A.1.4	Release policy and version numbers	591
A.1.5	Software requirements and backward compatibility	592
A.1.6	Terms of usage	593
A.2	Public data sets and test cases	594
A.3	More about modules and advanced functionality	596
A.3.1	Operating the module system	596
A.3.2	What characterizes a module?	597

A.3.3 List of modules	598
A.4 Rapid prototyping using MATLAB and MRST	605
A.5 Automatic differentiation in MRST	608
References	617

Preface

There are many 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; a comprehensive list can be found in the bibliography. However, neither of these books fully describe how you should implement these models and numerical methods to form a robust and efficient simulator. Some books 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. At the time of writing, the software has more than 12.000 unique downloads, more than 100 students have used the software in their master or PhD thesis, and approximately 150 papers written by authors outside of SINTEF include numerical experiments run in MRST. This book tries to give an in depth introduction to the two different programming paradigms that can be found in MRST. The

current version is up-to-date with respect to the latest development in data structures and syntax, both for the procedural parts of MRST that primarily focus on incompressible flow, as well as for the object-oriented, automatic-differentiation (AD-OO) framework that has been developed more recently to simulate compressible, multiphase flow.

The book has grown much longer than I anticipated when I started writing. Initially, my ambition was to provide introductory material on single-phase and two-phase incompressible flow, and discuss how such models could be implemented in a flexible and efficient manner by use of MATLAB. While I was writing, the software expanded rapidly and I had a hard time keeping pace. Inspired in part by the many people who have downloaded (and cited) the preliminary editions I have published online, but also as a result of numerous requests, I decided to also include a detailed discussion of the physics underlying industry-standard black-oil simulators and give a thorough introduction to how such models have been implemented in MRST. Now that this is done, it is time to stop. MRST has more to offer, including solvers for various EOR models, compositional flow, fractured media, geomechanics, geochemistry, as well as a comprehensive set of tools for modelling CO₂ storage in large-scale aquifer systems, but documentation of these will have to be another book or perhaps also another author. At the moment, I am more than happy that I finally managed to finish this book.

I would like to thank my current and former colleagues at SINTEF with whom I have collaborated over many years to develop MRST; primarily Olav Møyner, Halvor Møll Nilsen, Jostein R. Natvig, Stein Krogstad, Odd Andersen, Bård Skaflestad, and Xavier Raynaud. The chapter on flow diagnostics is the result of many discussions with Brad Mallison from Chevron. I am also grateful to the University of Bergen and the Norwegian University of Science and Technology for funding through my Professor II positions. Victor Calo and Yalchin Efendiev invited me to KAUST, where important parts of the chapters on grids and petrophysics were written. Likewise, Margot Gerritsen invited me to Stanford and gave me the opportunity to develop Jolts videos that complement the material in the book. Significant parts of the chapter on black-oil models was written during the three weeks I participated in a Long Program on Computational Issues in Oil Field Applications at UCLA. Last, but not least, I would like to thank colleagues and students who have given suggestions, pointed out errors and misprints, asked questions, and given me inspiration to continue working. Even though your name is not mentioned here, I have not forgotten all your important contributions.

Finally to the reader: 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 has neither been properly proofread or copy edited*. The text will therefore likely contain a great number of misprints, but hopefully fewer blatant errors. If you run across any, I would be grateful if you

point them out to me in an email and thereby help to improve the manuscript. Every now and then you may even encounter some text that has been **marked in dark red color** to indicate that it still needs editing. Suggestions for other improvement are also much welcome.

Oslo,
October 19, 2017

Knut-Andreas Lie
Knut-Andreas.Lie@sintef.no

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.

A third challenge is petroleum production. The civilized world will very likely continue to depend on the utilization of petroleum resources both as an energy carrier and as a raw material for consumer products in the foreseeable future. In recent years, conventional petroleum production has declined and the rate of new major discoveries has been significantly reduced: Optimal utilization of current fields and new discoveries is therefore 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. Herein, we will mainly focus on modelling flow in oil and gas reservoirs, which is often referred to as reservoir simulation. However, the general modelling framework and the numerical methods that are discussed also apply to modelling other types of flow in consolidated and saturated porous media.

In the book, we will introduce and discuss basic physical properties and mathematical models that are used to represent porous rocks and describe flow processes on a macroscopic scale. The presentation will focus primarily on physical processes that take place during hydrocarbon production. What this means is that even though the mathematical models, numerical methods, and software implementations presented can be applied to any of the applications outlined above, the specific examples use vocabulary, physical scales, and balances of driving forces that are specific to petroleum production. As an example of vocabulary, we can consider the ability of a porous medium to transmit fluids. In petroleum engineering this is typically given in terms of the 'permeability', which is a pure rock property, whereas one in water resource engineering is more concerned with the 'hydraulic conductivity' that also takes the viscosity and density of the fluid into account; in CO₂ sequestration you can see both quantities used. As an example of physical scales, let us compare oil production by water flooding and the question of long-term geological storage of CO₂. The hydrocarbons that make up petroleum resources can only accumulate when their natural upward movement relative to water is prevented by confinements in the overlying rocks, and hence the fluid flow in a petroleum reservoir takes place in a relatively closed system. Hydrocarbons will typically be produced for tens of years, during which the main driving mechanism is viscous forces induced by the pressure difference between the points where water is injected and oil is produced, which cause water to displace oil over distances of hundred to thousands of meters. Huge aquifer systems that stretch out for hundreds of kilometers are currently the most promising candidates for large-scale geological storage. During the injection phase, the flow processes of CO₂ storage are almost identical to those of petroleum production, albeit the operational constraints may differ, but as the CO₂ moves into the aquifer and the effects of the injection pressure ceases, the fluid movement will be dominated by buoyant forces that will cause the lighter CO₂ phase to migrate upward in the open aquifer system, and potentially continue to do so for thousands of years. In both cases, the governing equations of the basic flow physics are the same, but the balances between

physical forces are different, which should be accounted for when formulating the overall mathematical models and appropriate numerical methods.

Techniques developed to study 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 Petroleum production

To provide context for the discussion that will follow later in the book, we will briefly outline the various ways by which hydrocarbon can be produced from a subsurface reservoir. Good reservoir rocks have large void spaces between the mineral grains forming networks of connected pores that can store and transmit large amounts of fluids. Conceptually, one can think of a hydrocarbon reservoir as a bent, rigid sponge that is confined inside an insulating material and has all its pores filled with hydrocarbons that may appear in the form of oil or gas as illustrated in Figure 1.1. Natural gas will be dissolved in oil under high pressure like carbon-dioxide inside a soda can. If the pressure inside the pristine reservoir is above the bubble point, the oil is undersaturated and still able to dissolve more gas. If the pressure is below the bubble point, the oil will be fully saturated with gas and any excess gas will form a gas cap on top of the oil since it is lighter. To extract oil from the reservoir, one drills a well into the oil zone. The pristine pressure inside the reservoir may be sufficient to push hydrocarbons up to the surface. Alternatively, one may have to pump to lower the pressure beyond the point where oil starts flowing. How large the pressure differential needs to be for oil to flow will depend on the permeability of the rock; the higher the permeability is, the easier the hydrocarbons will flow towards the well.

As oil is extracted, the pressure inside the reservoir will decay and the production will gradually decline. However, declining pressure will often induce physical processes that contribute to maintain the production:

- In a *water drive*, the pore space below the hydrocarbons is filled with salt water that is slightly compressible, and hence will expand a little as the reservoir pressure is lowered. If the total water volume is large compared with the oil zone, even a small expansion will create significant water volumes that will push oil towards the well and hence contribute to maintain pressure. Sometimes the water is part of a large aquifer system that has a natural influx that replenishes the extracted oil by water and maintains pressure.

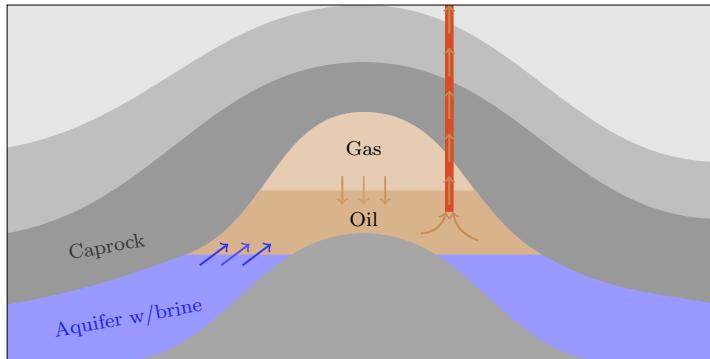


Fig. 1.1. Conceptual illustration of a petroleum reservoir during primary production. Over millions of years, hydrocarbons have accumulated under a caprock that has low ability to transmit fluids and therefore prevents their upward movement. Inside the trap, the fluids will be distributed according to density, with light gas on top, oil in the middle, and brine at the bottom. If the difference in pressure between the oil zone and the well is sufficiently high, the oil will flow naturally out of the reservoir. As oil is produced, the pressure inside the reservoir will decline, which in turn may introduce other mechanisms that contribute to maintain pressure and push more oil out of the well.

- *Solution gas drive* works like when you shake and open a soda can. Initially, the pristine oil will be in a pure liquid state and contain no free gas. The extraction of fluids will gradually lower the reservoir pressure below the bubble point, which causes free gas to develop and form expanding gas bubbles that force oil into the well. Inside the well, the gas bubble rise with the oil and make the combined fluid lighter and hence easier to push upward to the surface. At a certain point, however, the bubbles may reach a critical volume fraction and start to flow as a single gas phase that has lower viscosity than the oil and hence moves faster. This rapidly depletes the energy stored inside the reservoir and causes the production to falter. Gas coming out of solution can also migrate to the top of the structure and form a gas cap above the oil that pushes down on the liquid oil and hence contributes to maintain pressure.
- In a *gas cap drive*, the reservoir contains more gas than what can be dissolved in the oil. When pressure is lowered the gas cap expands and pushes oil into the well. Over time, the gas cap will gradually infiltrate the oil and cause the well to produce increasing amounts of gas.
- If a reservoir is highly permeable, gravity will force oil to move downward relative to gas and upward relative to water. This is called *gravity drive*.
- In a *combination drive* there is water below the oil zone and a gas cap above that both will push oil to the well at the same time as the reservoir pressure is reduced.

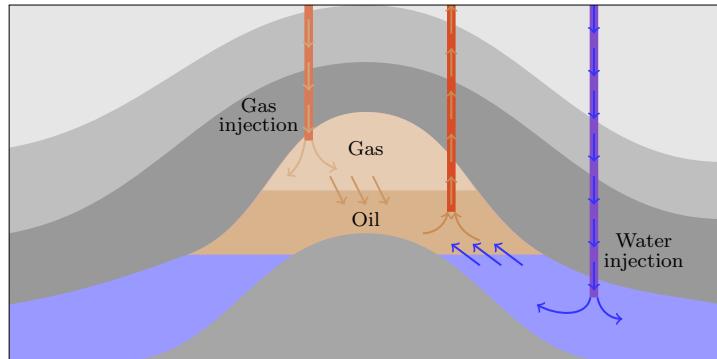


Fig. 1.2. Conceptual illustration of voidage replacement, which is an example of a secondary production strategy in which gas and/or water is injected to maintain the reservoir pressure.

These natural (or primary) drive mechanisms will only be able to maintain the pressure for a limited period and the production will gradually falter as the reservoir pressure declines. How fast the pressure declines and how much oil one can extract before the production ceases, varies with the drive mechanism. Solution gas drives can have a relatively rapid decline, whereas water and gas cap drives are able to maintain production for longer periods. Normally only 30% of the oil can be extracted using primary drive mechanisms.

To keep up the production and increase the recovery factor, most reservoirs will use some kind of engineered drive mechanisms. Figure 1.2 illustrates two examples of voidage replacement in which water and/or gas is injected to support pressure in the reservoir. Water can also be injected to sweep the reservoir, displace the oil, and push it towards the wells. In some cases, one may choose to inject produced formation water that is contaminated with hydrocarbons and solid particles and hence must be disposed of in some manner. Alternatively, or one can extract formation water from a nearby aquifer. In offshore production it is also common to inject seawater. A common problem for all waterflooding methods is to maximize the sweep efficiency so that water does not move rapidly through high-flow zones in the reservoir and leaves behind large volumes of unswept, mobile oil. Maintaining good sweep efficiency is particularly challenging for reservoirs containing high-viscosity oil. If injected water has low viscosity, it will tend to form viscous fingers that rapidly expand through the oil and cause early water breakthrough in the production wells. (Think of water being poured into a cup of honey). To improve the sweep efficiency, one can add polymers to the water to increase its viscosity and improve the mobility ratio between the injected and displaced fluid. Polymers have also been used to create flow diversions by plugging high-flow zones so that the injected fluid contacts and displaces more oil. For heavy

oils, adverse mobility ratios can be improved by using steam injection or some other thermal method to heat the oil to reduce its viscosity.

Water flooding, polymer injection, and steam injection are all examples of methods for so-called enhanced oil recovery (EOR). Another example is miscible and chemical injection, where one uses a solvent or surfactant that mixes with the oil in the reservoir to make it flow more readily. The solvent may be a gas such as carbon dioxide or nitrogen. However, the most common approach is to inject natural gas produced from the reservoir when there is no market that will accept the gas. Surfactants are similar to detergents used for laundry. Alkaline or caustic solutions, for instance, can react with organic acids occurring naturally in the reservoir to produce soap. The effect of all these substances is that they reduce the interfacial tension between water and oil, which enables small droplets of oil that were previously immobile to flow (more) freely. This is the same type of process that takes place when you use detergent to remove oily and greasy stains from textiles. A limiting factor of these methods is that the chemicals are quickly adsorbed and lost into the reservoir rock.

Often, one will want to combine methods that improve the sweep efficiency of mobile oil with methods that mobilize immobile oil. Miscible gas injection, for instance, can be used after a waterflood to flush out residually trapped oil and establish new pathways to the production wells. Water-alternating-gas (WAG) is the most successful and widely used EOR method. Injecting large volumes of gas is expensive, and by injecting alternating slugs of water, one reduces the injected volume of gas required to maintain pressure. Similarly, presence of mobile water reduces the tendency of the injected gas to finger through the less mobile oil. In polymer flooding, it is common to add surfactants to mobilize immobile oil by reducing or removing the interface tension between oil and water, and likewise, add alkaline solutions to reduce the adsorption of chemicals onto the rock faces.

While the mechanisms of all the above methods for enhanced oil recovery are reasonably well studied and understood, there are other methods whose mechanisms are much debated. This includes injection of low-salinity water, which is not well understood even though it has proved to be highly effective in certain cases. Another example is microbial enhanced oil recovery which relies on microbes that digest long hydrocarbon molecules to form biosurfactants or emit carbon dioxide that will reduce interfacial tension and mobilize immobile oil. Microbial activity can either be achieved by injecting bacterial cultures mixed with a food source, or by injecting nutrients that will activate microbes that already reside in the reservoir.

Use of secondary recovery mechanisms has been highly successful. On the Norwegian Continental Shelf, for instance, the average recovery factor is now almost 50%, which can be attributed mainly to water flooding and miscible gas injection. In other parts of the world, chemical methods have proved to be very efficient for onshore reservoirs having relatively short distances between wells. For offshore fields, however, the potential benefits of using chemical

methods are much debated. First of all, it is not obvious that such methods will be effective for reservoirs characterized by large inter-well distances as rapid adsorption onto the pore walls generally makes it difficult to transport the active ingredients long distances into a reservoir. Chemicals are also costly, need to be transported in large quantities, and may consume space on the platforms.

Even small improvements in recovery rates can lead to huge economic benefits for the owners of a petroleum asset and for this reason much research and engineering work is devoted to improve the understanding of mobilization and displacement mechanisms and to design improved methods for primary and enhanced oil recovery. Mathematical modeling and numerical reservoir simulation play key roles in this endeavor.

1.2 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 oil companies position and manage wells and well trajectories to maximize recovery of oil and gas. 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 discoveries 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 reservoirs¹ stretching out several square kilometers beneath remote and stormy seas, for which extensive simulation studies are inevitable to avoid making suboptimal and 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 that describe the relationship among different physical quantities. Second, one needs a geological

¹ The largest reservoir in the world is found in Ghawar in the Saudi Arabian desert and is approximately 230 km long, 30 km wide, and 90 m thick.

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 reason is that we can never get a complete and accurate characterization of the rock parameters that influence the flow pattern. 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 far exceed the capabilities of modern multi-processor 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 millions of 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 the 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 and mathematical models, as opposed to traditional methods that were developed for a large class of differential equations.

1.3 Outline of the book

The book is intended to serve several purposes. First of all, you can use the book as a self-contained introduction to the basic theory of flow in porous media and the numerical methods used to solve the underlying differential equations. Hopefully, the book will also give you a hands-on introduction to practical modeling of flow in porous media, focusing in particular on models and problems that are relevant to the petroleum industry. The discussion of mathematical models and numerical methods is accompanied by a large number of illustrative examples, ranging from idealized and highly simplified examples to cases involving models of real-life reservoirs.

The software aspect: user guide, examples, and exercises

All examples in the book have been created using the MATLAB Reservoir Simulation Toolbox (MRST), which we will discuss in more detail in Chapter A. MRST is an open-source software that can either be used as a set of gray-box reservoir simulators and workflow tools you can modify to suit your own purposes, or as a collection of flexible and efficient software libraries and data structures you can use to design your own simulators or computational workflows. The use of MRST permeates more traditional textbook material, and the book can therefore be seen as a user guide to MRST. Alternatively, the book can be viewed as a discussion by example of how a scripting language like MATLAB can be used for rapid prototyping, testing, and verification on realistic problems with a high degree of complexity. Through the many examples, we also try to gradually teach you some of the techniques and programming concepts that have been used to create MRST, which you can use to ensure flexibility and high efficiency in your own programs.

In the introductory part of the book that covers grids, petrophysical parameters and basic discretizations and solvers for single-phase flow, we have tried to make all code examples as self-contained as possible. To this end, we present and discuss in detail all code lines necessary to produce the numerical results and figures presented. However, sometimes we omit minor details that either have been discussed elsewhere or should be part of your basic MATLAB repertoire. As we move to more complex examples, in particular for multiphase flow and reservoir engineering workflows, it is no longer expedient to discuss MATLAB scripts in full details. In most cases, however, complete scripts that contain all code lines necessary to run the examples can be found in a dedicated `book` module that is part MRST. We strongly encourage you to use your own computer to run as many as possible of the examples in the book as well as other examples and tutorials that are distributed with the software. Your understanding will be further enhanced if you also modify the examples, e.g., by changing the input parameters, or extend them to solve problems that are related, but (slightly) different. To help you in this direction, we have included a number of computer exercises that modify and extend some of the

examples, combine ideas from two or more examples, or investigate in more detail aspects that are not covered by any of the worked examples. For some of the exercises you can find solution proposals in the `book` module.

Finally, we point out that MRST is an open-source software, and if reading this book gives you ideas about new functionality, or you discover things that are not working as they should or could, you are welcome to contribute to improve the toolbox and extend it in new directions.

Part I: geological models and grids

The first part of the book discusses how to represent a geological medium as a discrete computer model that can be used to study the flow of one or more fluid phases. Chapter 2 gives you a crash course in petroleum geology and geological modeling, written by a non-geologist. In this chapter, I try to explain the key geological processes that can lead to the formation of a hydrocarbon reservoir, discuss modeling of permeable rocks across multiple spatial scales, and introduce you to the basic physical properties that are used to describe porous media in general. For many purposes, reservoir geology can be represented as a collection of maps and surfaces. However, if the geological model is to be used as input to a macroscale fluid simulation, we must assume a continuum hypothesis and represent the reservoir in terms of a volumetric grid, in which each cell is equipped with a set of petrophysical properties. On top of this grid, one can then impose mathematical models that describe the macroscopic continuum physics of one or more fluid phases flowing through the microscopic network of pores and throats between mineral grains that are present on the subscale inside the porous rock of each grid block.

In Chapter 3, we describe in more detail how to represent and generate grids, with special emphasize on the types of grids that are commonly used in reservoir simulation. The chapter presents a wide variety of examples to illustrate and explain different grid formats, from simple structured grids, via unstructured grids based on Delaunay tessellations and Voronoi diagrams, to stratigraphic grids represented on the industry-standard corner-point format or as 2.5D unstructured grids. The examples also demonstrate various methods for generating grids that represent plausible reservoir models and discuss some of the realistic data sets that can be downloaded along with the MRST software.

Through Chapters 2 and 3, you will be introduced to the data structures MRST uses to represent grids and petrophysical data. Understanding these basic data structures, and the various methods that can be used to create and manipulate them, is fundamental if you want to understand the inner workings of a majority of the routines implemented in MRST or use the software as a platform to implement your own computational methods. Through the many examples, you will also be introduced to various functionality in MRST for plotting data associated with cells and faces (interface between two neighboring cells) as well as various strategies for traversing the grid and picking subsets of data that will prove very useful later in the book.

Part II: single-phase flow

The second part of the book is devoted entirely to single-phase flow and will introduce you to many of the key concepts for modeling flow in porous media, including basic flow equations, closure relationships, auxiliary conditions and models, spatial and temporal discretizations, and linear and nonlinear solvers.

Chapter 4 starts by introducing the two fundamental principles necessary to describe flow in porous medium, conservation of mass and Darcy's law, and outline different forms that these two equations can take in various special cases. Whereas mass conservation is a fundamental physical property, Darcy's law is phenomenological description of the conservation of momentum that uses permeability, a rock property, to relate volumetric flow rate to pressure differentials. To form a full model, the basic flow equations must be extended with various constitutive laws and extra equations describing the external forces that drive the fluid flow; these can either be boundary conditions and/or wells that inject or produce fluids. The last section of Chapter 4 introduces the classical two-point finite-volume method, which is the current industry-standard method used to discretize the mathematical flow equations. In particular, we demonstrate how to write this discretization in a very compact way by introducing discrete analogues of the divergence and gradient operators. These operators will be used extensively later when developing solvers for compressible single and multiphase flow.

Chapter 5 focuses on the special case of an incompressible fluid flowing in a completely rigid medium, for which the flow model can written as a Poisson-type partial differential equation (PDE) with a varying coefficient. We start by introducing the various data structures that are necessary to make a full simulator, including fluid properties, reservoir states describing the primary unknowns, fluid sources, boundary conditions, and models of injection and production models. We then discuss in detail the implementation of a two-point pressure solver, as well as upwind solvers for time-of-flight and tracer. We end the chapter by four simulation examples: the first introduces the classical quarter five-spot pattern, which is a standard test case in reservoir simulation, while the next three discuss how to incorporate boundary conditions and Peaceman well models and the difference between using structured and unstructured grids.

Grid used to describe real reservoirs typically have an unstructured topology and grid cells with irregular geometry and high aspect ratios. The standard two-point discretization is unfortunately consistent on such grids in special cases. To improve the spatial discretization, Chapter 6 introduces and discusses a few recent methods for consistent discretization on general polyhedral grids that are still being researched by academia. This includes mixed finite-element methods, multipoint flux approximation methods, and mimetic finite-difference methods.

Chapter 7, the last in Part II, is devoted to compressible flow, which in the general case is modelled by a nonlinear, time-dependent, parabolic PDE. Using

this relatively simple model, we introduce many of the concepts that will later be used to develop multiphase simulators of full industry-standard complexity. To discretize the transient flow equation, we combine the two-point method introduced for incompressible flow with an implicit temporal discretization. The standard approach for solving the nonlinear system of discrete equations arising from complex multiphase models is to compute the Jacobian matrix of first derivatives for the nonlinear system and use Newton's method to successively find a better approximations to the solution. Deriving and implementing analytic expressions for Jacobian matrices is both error-prone and time-consuming, in particular if the flow equations contain complex fluid model, well descriptions, thermodynamical behavior, etc. In MRST, we have therefore chosen to construct Jacobian matrices using automatic differentiation, which is a technique to numerically evaluate the derivatives of functions specified by a computer program to working precision accuracy. Combining this technique with discrete averaging and differential operators enables you to write very compact simulator codes in which the flow models are implemented almost in the same form as they are written in the underlying mathematical equations. This opens up for a simple way of writing new simulators: all you have to do is to implement the new model equations in residual form and specify which variables should be used in the linearization of the resulting nonlinear system. Then the software computes the corresponding derivatives and assembles them into a correct block matrix. To demonstrate the utility and power of the resulting framework, we show how one can quickly change functional dependencies in the single-phase pressure solver and extend it to include new functional dependencies, thermal effect, and non-Newtonian fluid rheology.

Part III: multiphase flow

The third part of the book outlines how to extend the ideas from Part II to multiphase flow. Chapter 8 starts by introducing new physical phenomena and properties that appear for multiphase flows, including fluid saturations, wettability and capillary pressure, relative permeability, etc. With this introduced, we move on to outline the general flow equations for multiphase flow, before we discuss various model reformulations and (semi-)analytical solutions for the special case of immiscible, two-phase flow. The main difference between single and multiphase flow is that we now, in addition to an equation for fluid pressure (or fluid density), get additional equation for the transport of fluid phases and/or chemical species. These equations are generally parabolic, but will often behave like, or simplify to, hyperbolic equations. Chapter 9 therefore introduces basic concept from the theory of hyperbolic conservation laws and introduces various numerical schemes, from classical to modern high-resolution schemes. We also introduce the basic discretization that will be used for transport equations later in the book.

The next chapter follows along the same lines as Chapter 5 and explains how the incompressible solvers developed for single-phase flow can be extended

to account for multiphase flow effects using the so-called fractional-flow formulation introduced in Chapter 8 and simulated using sequential methods in which pressure effects and transport of fluid saturations and/or component concentrations are computed in separate steps. The chapter includes a number of test cases that discuss various effect of mulitphase flow and illustrates error mechanisms inherent in sequential simulation methods.

Once the basics of incompressible, mulitphase flow has been discussed, we move on to discuss more advanced multiphase flow models, focusing primarily on the black-oil formulation which can be found in contemporary commercial simulators. Chapter 11 gives an in-depth discussion of the physical principles of such models, whereas Chapter 12 discusses how to one can develop robust simulators by use of variants of the automatic-differentiation methods introduced in Chapter 7 combined with modern principles of object orientation. The two chapters also include a number of illustrative simulation cases.

Part IV: reservoir engineering workflows

The fourth, and last part of the book, is devoted to discussing additional computational methods and tools that are commonly used in reservoir engineering workflows.

Chapter 13 is devoted to flow diagnostics, which are simple numerical experiments that can be used to probe a reservoir to understand flow paths and communication patterns. Herein, all types of flow diagnostics will be based on the computation of time-of-flight, which defines natural time lines in the porous medium, and steady state distribution of numerical tracers, which delineate the reservoir into sub-regions that can be uniquely associated with distinct parts of the inflow/outflow boundaries. Both these quantities will be computed using finite-volume method introduced in Chapters 4 and 5. The concept of flow diagnostics also includes several measures of dynamic heterogeneity, which can be used as simple proxies for more comprehensive multiphase simulations in various reservoir engineering workflows including placement of wells, rate optimization, etc.

As you will see in Chapter 2, porous rocks are heterogeneous at a large variety of length scales. There is therefore a general trend to build complex, high-resolution models for geological characterization to represent small-scale geological structures. Likewise, large ensembles of equiprobable models are routinely generated to systematically quantify model uncertainty. In addition, many companies develop hierarchies of models that cover a wide range of physical scales to systematically propagate the effects of small-scale geological variations up to the reservoir scale. In either case, one quickly ends up with geological models that contain more details than what can be used, or should be used, in subsequent flow simulation studies. Hence, there is a strong need for mathematical and numerical techniques that can be used to develop reduced models or to communicate effective parameters and properties between models of different spatial resolution. Such methods are discussed in Chapters 14 and

15. In Chapter 14 we introduce data structures and various methods that can be used to partition a fine-scale grid into a coarse-scale grid. Chapter 15 then discusses upscaling, which refers to the process in which petrophysical properties in the cells that make up a coarse block are averaged into a single effective value for each coarse block.

Having been introduced to the key aspects of the book, you should be ready to start digging into the material. Before continuing to the next chapter, we present a simple example that will give you a first taste of simulating flow in porous media.

1.4 The first encounter with MRST

The purpose of this first example is to show the basic steps for setting up, solving, and visualizing a simple flow problem using MRST. 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

$$\nabla \cdot \vec{v} = 0, \quad \vec{v} = -\frac{K}{\mu} [\nabla p + \rho g \nabla z], \quad (1.1)$$

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 millidarcy (md or mD), whereas the porosity (i.e., the void fraction of the bulk volume) is set to 0.2. The fluid has a density ρ of 1000 kg/m³ and viscosity μ equal one centipoise (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, $[0, 1] \times [0, 1] \times [0, 30]$ m³, which we discretize using a regular $1 \times 1 \times 30$ Cartesian grid. The simulation model is set up by constructing a grid and assigning the rock permeability, and setting boundary conditions:

```
gravity reset on
G    = cartGrid([1, 1, 30], [1, 1, 30]*meter^3);
G    = computeGeometry(G);
rock = makeRock(G, 0.1*darcy(), 0.2);
```

MRST works in SI units and we must therefore 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, that relates the flux between cells to their

pressure difference, $v_{ij} = -T_{ij}(p_i - p_j)$. For Cartesian grids, this scheme coincides with the classical seven-point scheme for Poisson's problem and is the only discretization that is available in the basic parts of MRST. More advanced discretizations can be found in the add-on modules. To define the two-point discretization, we compute therefore compute the transmissibilities T_{ij} , which can be defined independent of the particular flow model once we have defined the grid and petrophysical parameters:

```
T = computeTrans(G, rock);
```

Since we are solving (1.1) on a finite domain, we must also 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:

```
bc = pside([], G, 'TOP', 100.*barsa());
```

The next thing we need to define is the fluid properties. Unlike grids, petrophysical data, and boundary conditions, data structures for representing fluid properties are not part of the basic functionality of MRST. The reason is that the way fluid properties are specified is tightly coupled with the mathematical and numerical formulation of the flow equations, and may differ a lot between different types of simulators. Here, we have assumed incompressible flow and can therefore use fluid models from the `incomp` add-on module,

```
mrstModule add incomp;
fluid = initSingleFluid('mu' , 1*centi*poise, ...
'rho', 1014*kilogram/meter^3);
```

As a final step, we use the transmissibilities, the fluid object, and the boundary conditions to assemble and solve the discrete system:

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

Having computed the solution, we plot the pressure given in units 'bar', which equals 0.1 MPa and is referred to as 'barsa' in MRST since 'bar' is a built-in command in MATLAB:

```
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])
```

From the plot shown in Figure 1.3, we see that our solution correctly reproduces the linear pressure increase with depth one would expect to find inside a column consisting of a single fluid phase.

At this point, I encourage you to consult Chapter A which describes the MATLAB Reservoir Simulation Toolbox in more detail. Knowing your way around MRST is not a prerequisite for reading what follows, but will definitely contribute significantly to increase your understanding. If you are already

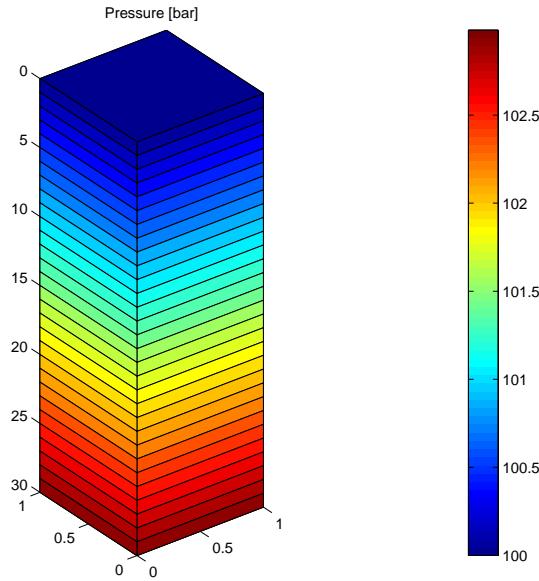


Fig. 1.3. Hydrostatic pressure distribution in a gravity column computed by MRST. This example is taken from the MRST tutorial `flowSolverTutorial1.m` (`gravityColumn.m` in older versions of MRST).

familiar with the software, or want to postpone the introduction, you can continue directly to Chapter 2.

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 briefly how sedimentary rocks and hydrocarbons are formed. In doing so, we introduce some geological concepts that you may encounter while working with subsurface modelling. We then move on to describe how rocks that contain hydrocarbons or aquifer systems are modelled. Finally, we discuss how rock properties are represented in MRST and show several examples of rock models with varying complexity, ranging from an idealized shoe-box rock body with homogeneous properties, via the widely used SPE 10 model, to two realistic models, one synthetic and one representing a large-scale aquifer from the North Sea.

2.1 Formation of sedimentary rocks

Sedimentary rocks are created by mineral or organic particles that are deposited and accumulated on the Earth's surface or within bodies of water to create layer upon layer of sediments. The sedimentary rocks that are found in reservoirs come from sedimentary basins, inside which large-scale sedimentation processes have taken place. Sedimentary basis are formed as the result of stretching and breaking of the continental crust. As the crust is stretched, hot rocks deeper in the earth come closer to the surface. When the stretching stops, the hot rocks start to cool, which causes the crustal rock to gradually subside and move downward to create a basin. Such processes are also taking place today. The Great Rift Valley of Africa is a good example of a so-called rift basin, where a rift splits the continental plate so that opposite sides of the valley are moving a millimeter apart each year. This gradually creates a basin inside which a new ocean may appear over the next hundred million years.

On the part of the Earth's surface that lies above sea level, wind and flowing water will remove soil and rock from the crust and transport it to another place where it is deposited when the forces that transport the mineral particles can no longer overcome gravity and friction. Mineral particles can also be transported and deposited by other geophysical processes like mass movement and glaciers. Over millions of years, layers of sediments will build up in deep waters, in shallow marine-waters along the coastline, or on land in lakes, rivers, sand deltas, and lagoons, see Figure 2.1. The sediments accumulate a few centimeters every hundred years to form sedimentary beds (or strata) that may extend many kilometers in the lateral directions.

Over time, the weight of the upper layers of accumulating sediment will push the lower layers downward. A combination of heat from the Earth's center and pressure from the overburden will cause the sediments to undergo various chemical, physical, and biological processes (commonly referred to as diagenesis). These processes cause the sediments to consolidate so that the loose materials form a compact, solid substance that may come in varying forms. This process is called lithification. Sedimentary rocks have a layered structure with different mixtures of rock types with varying grain size, mineral types, and clay content. The composition of a rock depends strongly upon a combination of geological processes, the type of sediments that are transported, and the environment it is deposited. Sandstone are formed by mineral particles that are broken off from a solid material by weathering and erosion in a source area and transported by water to a place where they settle and accumulate. Limestone is formed by the skeletons of marine organisms like corals that have a high calcium content. Sandstone and limestone will typically contain relative large void spaces between mineral particles in which fluids can move easily. Mudrocks, on the other hand, are formed by compression of clay, mud, and silt and will consist of relatively fine-grained particles. These rocks, which are sometimes also referred to as shale, will have small pores and therefore have limited ability to transmit fluids. Chemical rocks are formed by minerals that precipitate from a solution, e.g., salts that are left behind when oceans evaporate. Like mudrocks, salts are impermeable to fluids. Sediments will also generally contain organic particles that originate from the remains of plants, living creatures, and small organisms living in water.

To understand a particular rock formation, one must understand the prehistoric sedimentary environment from which it originates. It is common to distinguish between continental and marine environments. The primary source of sediments in a continental environment is rivers, inside which the moving water has high energy so that sediments will mainly be composed of fragments of preexisting minerals and rock, called clasts. Resulting rocks are therefore often referred to as clastic rocks. A flood plain is the area of the land adjacent to the river bank that will be covered by water when a river breaks its bank and floods during periods of high water discharge. Flood plains are created by bends in the river (meanders) that erode sideways, see Figure 2.2. A flood plain is the natural place for a river to diminish its kinetic energy. As the

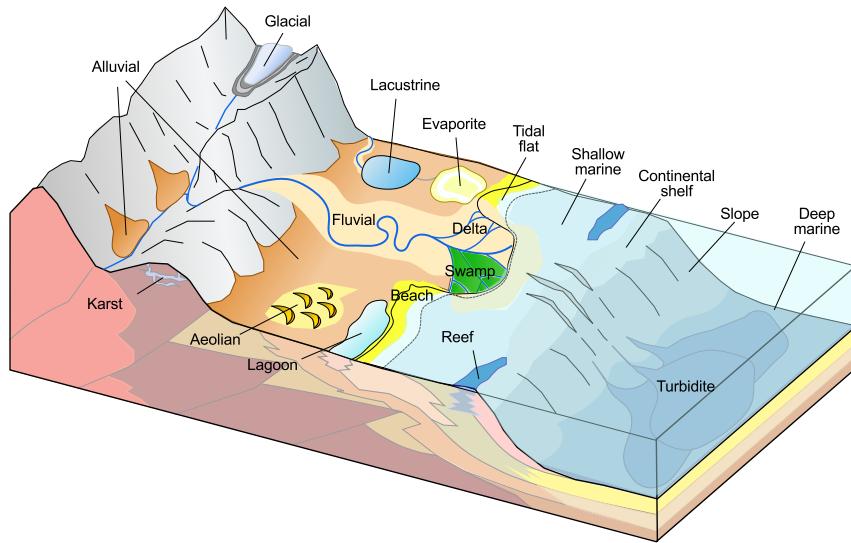


Fig. 2.1. Illustration of various forms of depositional environments. Aeolian sediments are created by wind, evaporite minerals are left behind as water evaporates, fluvial sediments are deposited by rivers, lacustrine sediments are deposited in lakes, and alluvial refers to the case where the geological process is not well described. The illustration is based on an image by PePeEfe on Wikimedia Commons: http://commons.wikimedia.org/wiki/File:Principales_medios_sedimentarios.svg. The modified image is published under the Creative Commons Attribution-Share Alike 3.0 Unported license.

water moves, particles of rock are carried along by frictional drag of water on the particle, and water must flow at a certain velocity to suspend and transport a particle. Large particles are therefore deposited where the water flows rapidly, whereas finer particles will settle where the current is weak. River systems also create alluvial fans, which are fan-shaped sediment deposits built up by streams that carry debris from a single source. Other examples of continental environments are lagoons, lakes, and swamps that contain quite water in which fine-grained sediments mingled with organic material are deposited.

For marine rocks formed in a sea or ocean, one usually makes the distinction between deep and shallow environments. In deep waters (200 meters or more), the water moves relatively slowly over the bottom and deposits will mainly consist of fine clay and skeletons of small microorganisms. However, if the sea bottom is slightly inclined, the sediments can become unstable and induce sudden currents of sediment-laden water to move rapidly down-slope. These so-called turbidity currents are caused by density differences between the suspended sediments and the surrounding water and will generally be highly turbulent. Because of the high density, a turbidity current can trans-

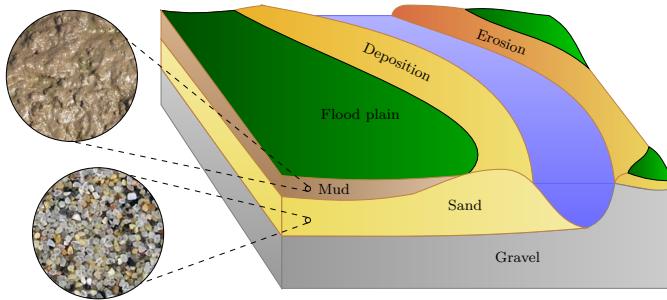


Fig. 2.2. Illustration of a meandering river. Because water flows faster on the outer than on the inner curve of the bend, the river will erode along the outer bank and deposit at the inner bank. This causes the river to move sideways over time. Occasionally, the river will overflow its banks and cover the lower-lying flood plain with water and mud deposits that makes the soil fertile. Flood plains are therefore very important for agriculture.

port larger particles than what pure water would be able to at the same velocity. These currents will therefore cause what can be considered as instantaneous deposits of large amounts of clastic sediments into deep oceans. The resulting rocks are called turbidite.

Shallow-marine environments are found near the coastline and contain larger kinetic energy caused by wave activity. Clastic sediments will therefore generally be coarser than in deep environments and will consist of small-grained sand, clay, and silt that has been washed out and transported from areas of higher water energy on the continent. Far from the continent, the transport of clastic sediments is small and deposits are dominated by biological activity. In warm waters, there are multitudes of small organisms that build carbonate skeletons, and when deposited together with mud, these skeletons will turn into limestone, which is an example of a carbonate rock. Carbonate rocks can also be made from the skeletons of larger organisms living on coral reefs. Carbonates are soluble in slightly acidic water, and this may create karst rocks that contain large regions of void space (caves, caverns, etc).

To model sedimentary rocks in a simulation model, one must be able to describe the geometry and the physical properties of different rock layers. 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 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 2.3 shows two photos of



Fig. 2.3. Outcrops of sedimentary rocks from Svalbard, Norway. The length scale is a few hundred meters.



Fig. 2.4. 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.

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 2.4 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 depositional 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. Simultaneously, severe geological activity takes place: Cracking of continental plates and volcanic activity will change what is to become

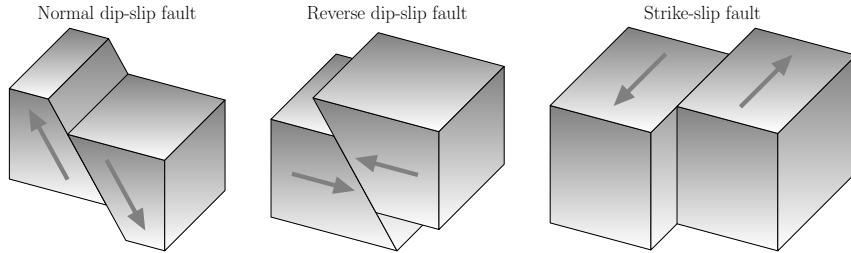


Fig. 2.5. Different types of faults: In strike-slip faults, the rock volumes slide past each other to the left and right in the lateral direction with very little vertical motion. In dip-slip faults, the rock volumes move predominantly in the vertical direction. A normal dip-slip fault occurs when the crust is extended and is sometimes called a divergent fault. Reverse dip-slip faults occur as a result of compressive shortening of the crust and is sometimes called convergent faults since rock volumes opposite sides of the fault plane move horizontally towards one another.

our reservoir from being a relatively smooth, layered sedimentary basin into a complex structure. Previously continuous layers of sediments are compressed and pushed against each other to form arches, which are referred to as anticlines, and depressions, which are referred to as synclines. If the deposits contain large accumulations of salts, these will tend to flow upward through the surrounding layers to form large domes since salts are lighter than other mineral particles. Likewise, as the sediments may be stretched, cut, shifted, or twisted in various directions, their movement may introduce fractures and faults. Fractures are cracks or breakage in the rock, across which there has been no movement. Faults are fractures or discontinuities in a volume of rock, across which movement in the crust has caused rock volumes on opposite sides to be displaced relative to each other. Some faults are small and localized, whereas others are part of the vast system of boundaries between tectonic plates that crisscrosses the crust of the Earth. Faults are described in terms of their strike, which is the compass direction in which the fault intersects the horizontal plane, and by their dip, which is the angle of the fault plane makes with the horizontal, measured perpendicular to the strike. Faults are further classified by their slip, which is the displacement vector that describes the relative movement of rock volumes on opposite sides of the fault plane. The dip is usually separated into its vertical component, called throw, and its horizontal component, called heave. Figure 2.5 illustrates different types of faults.

2.2 Creation of crude oil and natural gas

Deposits not only consist of sand grains, mud, and small rock particles but will also contain remains of plankton, algae, and other organisms living in

water that die and fall down to the bottom where they are covered in mud. During their life, these organisms have absorbed heat from the sunlight and when they die and mix with the sediments, they take energy with them. As the sediments get buried deeper and deeper, the increasing heat from the Earth's core and pressure from the overburden will compress and 'cook' the organic material that consists of cellulose, fatty oils, proteins, starches, sugar, waxes, and so on into an intermediate waxy product called kerogen. Whereas organic material contains carbon, hydrogen, and oxygen, the kerogen contains less oxygen and has a higher ratio of hydrogen to carbon. The maturation process that eventually turns kerogen into crude oil and natural gas depends highly upon temperature, and may take from a million years at 170° C to a hundred million of years at 100° C. Whereas most of the oil and natural gas we extract today has been formed from the remains of prehistoric algae and zooplankton living in the ocean, coal is formed from the remains of dead plants. The rock in which this 'cooking' process takes place is commonly referred to as the source rock. The chances of forming a source rock containing a significant amount of oil and gas increases if there has been an event that caused mass-death of microorganisms in an ocean basin.

Pressure from sediments lying above the source rock will force the hydrocarbons to migrate upward through the void space in these newer sediments. The lightest hydrocarbons (methane and ethane) that form natural gas usually escape quickly, whilst the liquid oils move more slowly towards the surface. In this process, the natural gas will separate from the oil, and the oil will separate from the resident brine (salty water). At certain sites, the migrating hydrocarbons will be collected and trapped in structural or stratigraphic traps. Structural traps are created by tectonic activity that forms layers of sediments into anticlines, domes, and folds. 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 layer such as mudstone, which consists of small and densely packed particles and thus has strong capillary forces that cannot easily be overcome by the buoyancy forces that drive migrating hydrocarbons upward. If a layer of mudstone is folded into an anticline above a more permeable sandstone or lime stone, it will therefore act as a caprock that prevents the upward migration of hydrocarbons. Stratigraphic traps are especially important in the search for hydrocarbons; approximately 4/5 of the world's oil and gas reserves are found in anticlinal traps. Figure 2.6 illustrates the various forms in which migrating oil and natural gas can be trapped and accumulate to form petroleum reservoirs.

The first evidence of hydrocarbons beneath the earth's surface were found in so-called seeps, which are found in many areas throughout the world. Seeps are formed when the seal above a trap is breached so that oil and gas can migrate all the way through the geological layers and escape to the surface. As the hydrocarbons break through the surface, the lighter components will continue to escape to the atmosphere and leave behind heavier products like bitumen, pitch, asphalt and tar have been exploited by mankind since pale-

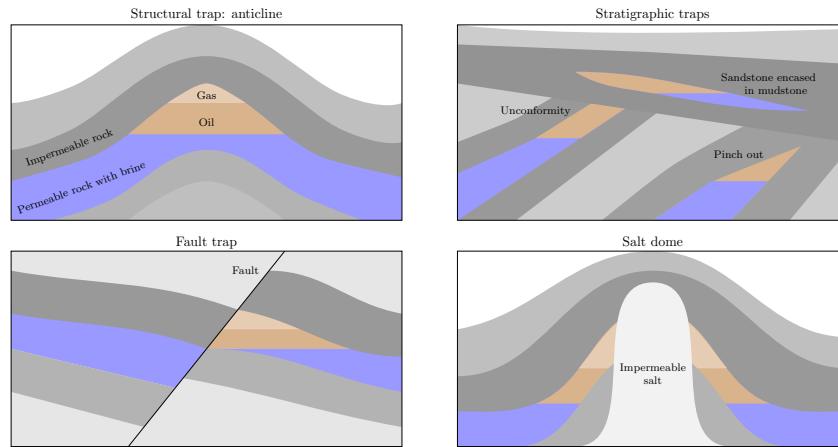


Fig. 2.6. Various forms of traps in which migrating hydrocarbons can accumulate. **Anticlines** are typically produced when lateral pressure causes strata to fold, but can also result from draping and subsequent compaction of sediments accumulating over local elevations in the topography. **Stratigraphic traps** accumulate hydrocarbons because of changes in the rock type. One example is sand banks deposited by meandering river that at later times is covered by mud from the flood plain. Similarly, stratigraphic traps can form when rock layers are tilted and eroded away and subsequently covered by other low-permeable strata. **Fault traps** are formed when strata are moved in opposite directions along a fault line so that permeable rocks come in contact with impermeable rocks. The fault itself can also be an effective trap if it contains clays that are smeared as the layers of rock move past each other. **Salt domes** are created by buried salt deposits that rise unevenly through the surrounding strata. Oil can either rest against the salt itself, or the salt induces chemical reactions in the surrounding rock that makes it impermeable.

olistic times. In 1859, Edwin L. Drake, also known as Colonel Drake, drilled the world's first exploration well into an anticline in Titusville, Pennsylvania to find oil 69.5 ft below the surface. Since then, an enormous amount of wells have been drilled to extract oil from onshore reservoirs in the Middle-East, North America, etc. Drilling of submerged oil wells started just before the turn of the 19th century: from platforms built on piles in the fresh waters of the Grand Lake St. Mary (Ohio) in 1891, and from piers extending into the salt water of the Santa Barbara Channel (California) in 1896. Today, hydrocarbons are recovered from offshore reservoirs that are located thousands of meters below the sea bed and in waters with a depth up to 2600 meters in the Gulf of Mexico, the North Sea, and offshore of Brazil.

2.3 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 micrometer scale of pore channels between the solid particles making up the rock to the kilometer 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, which are the smallest stratigraphic units. The thickness of beds varies from millimeters 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 kilometers.

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 2.12. 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 2.7. 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 (~ 50 cm), to the macroscopic facies association scale (~ 100 m) of geological models. In this book, we will primarily focus on another scale, simulation models, which represent

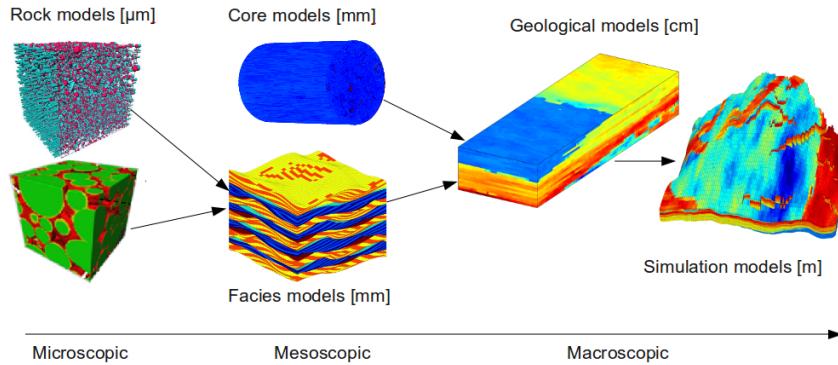


Fig. 2.7. Illustration of the hierarchy of flow models used in subsurface modeling. The length scales are the vertical sizes of typical elements in the models.

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.

2.3.1 Macroscopic models

To model a reservoir on a macroscopic scale, we basically need to represent its geology at a level of detail that is sufficient for the purpose the model is built to serve: to visualize how different experts perceive the reservoir, to provide estimates of hydrocarbon volumes, to assist well planning and geosteering, or as input to geophysical analysis (seismic modeling, rock mechanics) or flow simulations. For flow simulation, which is our primary concern in this book, we need a volumetric description that decomposes the reservoir into a set of grid cells (small 3D polygonal volumes) that are petrophysically and/or geometrically distinct from each other. With a slight abuse of terminology, we will in the following refer to this as the *geological model*, which we will distinguish from the models that describe the reservoir fluids and the forces that cause their movement.

Geological models are generally built in a sequence of steps, using a combination of stratigraphy (the study of rock layers and layering), sedimentology (study of sedimentary rocks), structural geology (the study of how rock layers are deformed over time by geological activity), diagenesis (the study of chemical, physical, and biological processes that transform sediments to rock), and interpretation of measured data. The starting point is usually a seismic interpretation, from which one obtains a representation of faults and geological horizons that bound different geological units. The seismic interpretation is used alongside a conceptual model in which geologists express how they believe the reservoir looks like based on studies of geological history

and geological outcrops. The result can be expressed as a geometric model that consists of vertical or inclined surfaces representing faults and horizontal or slightly sloping surfaces representing horizons that subdivide the reservoir volume into different geological units (zones). This zonation is obtained by combining seismic interpretation that describes the gross geometry of the reservoir with stratigraphic modelling and thickness information (isochores) obtain from well logs that defines the internal layering. Once a model of the structural and stratigraphic architecture of the reservoir is established, the surface description can be turned into a 3D grid that is populated with cell and face properties that reflect the geological framework.

Unfortunately, building a geological model for a reservoir is like finishing a puzzle where most of the pieces are missing. The amount of data available is limited due to the costs of acquiring them, and the data that is obtained is measured on scales that may be quite disparate from the geological features one needs to model. Seismic surveys give a sort of X-ray image of the reservoir, but 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. Information on finer scales is available from various measuring tools lowered into the wells to gather information of the rock in near-well region, e.g., by radiating the reservoir and measuring the response. Well-logs have quite limited resolution, rarely down to centimeter scale. The most detailed information is available from rock samples (cores) extracted from the well. The industry uses X-ray, CT-scan, as well as electron microscopes to gather high resolution information from the cores, and the data resolution is only limited by the apparatus at hand. However, information from cores and well-logs can only tell you how the rock looks like near the well, and extrapolating this information to the rest of the reservoir is subject to great uncertainty. Moreover, due to high costs, one 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 millimeters to several kilometers.

In summary, the process of making a geological model is generally strongly under-determined. It is therefore customary to use a combination of deterministic and probabilistic modeling to estimate the subsurface characteristics between the wells. Deterministic modeling is used to specify large-scale structures such as faults, correlation, trends, and layering, which are used as input and controls to geostatistical techniques that build detailed grid models satisfying statistical properties assumed for the petrophysical heterogeneity. Since trends and heterogeneity in petrophysical properties depend strongly on the structure of sedimentary deposits, high-resolution petrophysical realizations are in many cases not built directly. Instead, one starts by building a *rock model* that is based on the structural model and consists of a set of discrete rock bodies (facies) that are specified to improve petrophysical classification

and spatial shape. For carbonates, the modelled facies are highly related to diagenesis, while the facies modelled for sandstone reservoirs are typically derived from the depositional facies. 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.

To populate the modelled facies with properties it is common to use stochastic simulation techniques that simulate multiple realizations of the petrophysical properties represented as cell and face properties in high-resolution grid models. 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 2.4 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.

2.3.2 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 2.8. 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 kilometer-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

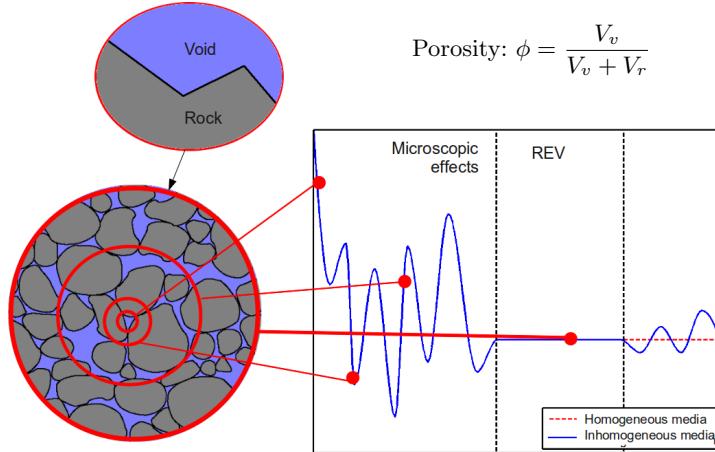


Fig. 2.8. The concept of a representative elementary volume (REV), here illustrated for porosity which measures the fraction of void space to bulk volume.

extending throughout the entire horizontal length of the reservoirs. Each of these individual layers may be nearly homogeneous because they are created 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.

2.3.3 Microscopic models: The pore scale

Pore-scale models, as illustrated to the left in Figure 2.7, 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 2.9. The main methods for obtaining pore-scale models from a rock sample is by studying thin slices using an electron microscope with micrometer



Fig. 2.9. Three core plugs with diameter of one and a half inches, and a height of five centimeters.

resolution or by CT-scans. In the following, we will give a simplified overview of flow modelling on this scale.

At the pore scale, the porous medium is either represented by a volumetric grid or by a graph (see e.g., [205]). 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 than 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., [205] 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).

2.3.4 Mesoscopic models

Models based on flow experiments on core plugs are by far the most common mesoscopic models. The main 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 1.4. 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 2.7. 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.

2.4 Modelling rock properties

Describing 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 2.3.3 or the through individual lamina as in Section 2.3.4. 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 2.3.2. 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.

2.4.1 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 rhomboidal packing to 0.4764 for cubic packing. When sediments are first deposited in water, they usually have a porosity of approximately 0.5, but as they are buried, the water is gradually squeezed out and the void space between the mineral particles decreases as the sediments are consolidated into rocks. For sandstone and limestone, ϕ is in the range 0.05–0.5, 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. Increase compaction (and cementation) causes porosity to decrease with depth in sedimentary rocks. The porosity can also be reduced by minerals that are deposited as water moves through the pore spaces. For sandstone, the loss in porosity is small, whereas shales loose their porosity very quickly. Shales are therefore unlikely to be good reservoir rocks, and will instead act like caprocks having porosities that

are orders of magnitude lower than those found in good sandstone and carbonates.

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

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

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

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

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

$$\phi = \phi_0 [1 + c_r(p - p_0)]. \quad (2.3)$$

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.

2.4.2 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 $\nabla\Phi$,

$$\vec{u} = -\frac{K}{\mu} \nabla\Phi. \quad (2.4)$$

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 [67]. 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. This 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 4.1.

The SI-unit for permeability is m^2 , which reflects the fact that permeability is determined entirely by the geometric characteristics of the pores. However, it is more common to use the unit 'darcy' (D). The precise definition of $1\text{D} \approx 0.987 \cdot 10^{-12} \text{ m}^2$ involves transmission of a 1 cP fluid through

a homogeneous rock at a speed of 1 cm/s due to a pressure gradient of 1 atm/cm. Translated to reservoir conditions, 1 D is a relatively high permeability and it is therefore customary to specify permeabilities in millidarcys (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 gases. 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 μ D or lower. 'Tight' reservoirs are defined as those having permeability less than 0.1 mD. 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 (2.4), we tacitly assumed that the permeability K is a scalar quantity. However, the permeability will generally be a full tensor,

$$\mathbf{K} = \begin{bmatrix} K_{xx} & K_{xy} & K_{xz} \\ K_{yx} & K_{yy} & K_{yz} \\ K_{zx} & K_{zy} & K_{zz} \end{bmatrix}. \quad (2.5)$$

Here, the diagonal terms $\{K_{xx}, K_{yy}, K_{zz}\}$ represent how the flow rate in one axial direction depends on the pressure drop in the same direction. The off-diagonal terms $\{K_{xy}, K_{xz}, K_{yx}, K_{yz}, K_{zx}, K_{zy}\}$ account for dependence between flow rate in one axial direction and the pressure drop in perpendicular directions. A full tensor is needed to model local flow in directions at an angle to the coordinate axes. Let us for instance consider a layered system, for which the dominant direction of flow will generally be along the layers. However, if the layers form an angle to the coordinate axes, a pressure drop in one coordinate direction will produce flow at an angle to this direction. This type of flow can only be modelled correctly with a permeability tensor with nonzero off-diagonal terms. If the permeability can be represented by a scalar function $K(\vec{x})$, we say that the permeability is *isotropic* as opposed to the *anisotropic* case where we need a full tensor $\mathbf{K}(\vec{x})$. To model a physical system, the anisotropic permeability tensor must be symmetric because of the Onsager principle of reciprocal relations and positive definite because the flow component parallel to the pressure drop should be in the same direction as the pressure drop. As a result, a full-tensor permeability \mathbf{K} may be diagonalized by a change of basis.

Since the porous medium is formed by deposition of sediments over thousands of years, there is often a significant difference between permeability in the vertical and lateral directions, but no difference between the permeabilities in the two lateral directions. The permeability is obviously also a function of

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

$$K = \frac{1}{8\tau A_v^2} \frac{\phi^3}{(1 - \phi)^2}, \quad (2.6)$$

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 (2.6) 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 2.5.

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 8.

2.4.3 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.

2.5 Property 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

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

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.

2.5.1 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 = makeRock(G, 200*milli*darcy, 0.2);
```

Because MRST works in SI units, it is important to convert from the field units 'darcy' to the SI unit 'meters²'. 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 = makeRock(G, [100 100 10].*milli*darcy, 0.2);
```

2.5.2 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 [71] 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 $d_p = 10 \mu\text{m}$, for which the specific surface area is $A_v = 6/d_p$. Using the Carman-Kozeny relation (2.6), 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.

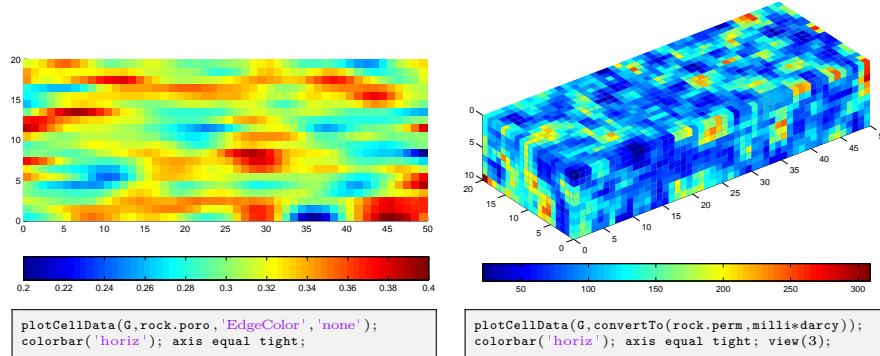


Fig. 2.10. The left plot shows a 50×20 porosity field generated as a Gaussian field with a larger filter size in x -direction than in the y -direction. The right plot shows the permeability field computed from the Carman-Kozeny relation for a similar $50 \times 20 \times 10$ porosity realization computed with filter size [3, 3, 3].

```

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 = makeRock(G, K(:, :, p(:)));

```

The resulting porosity field is shown in the left plot of Figure 2.10. 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 of different thickness. From top to bottom (stratigraphic grids are often numbered from the top and downward), the first layer is one cell thick and has a mean permeability value of 100 mD, the second layer is three cells thick and has mean permeability of 400 mD, the third layer is two cells thick and has mean value 50 mD, and the fourth layer is four cells thick and has mean value 350 mD. To specify this, we have used an indirection map. That is, if K_m is the n -vector of mean permeabilities and L is the $(n+1)$ -vector of indices, the value $K_m(i)$ is assigned to vertical layers number $L(i)$ to $L(i+1)-1$. The resulting permeability is shown in Figure 2.11.

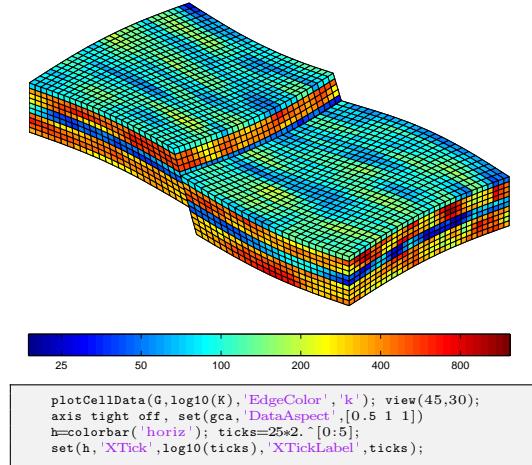


Fig. 2.11. A stratigraphic grid with a single fault and four geological layers, each with a lognormal permeability distribution.

2.5.3 10th SPE Comparative Solution Project: Model 2

Society of Petroleum Engineers (SPE) has developed a series of benchmarks that can be used to independently compare computational methods and simulators. The first nine benchmarks focus on black-oil, compositional, dual-porosity, thermal, and miscible simulations, as well as horizontal wells and gridding techniques. The 10th SPE Comparative Solution Project [62] was posed as a benchmark for upscaling methods, but the second data set of this benchmark has later become very popular within the academic community as a benchmark for comparing different computational methods. The data set is a 3-D geostatistical realization from the Jurassic Upper Brent formations, in which one can find the giant North Sea fields of Statfjord, Gullfaks, Oseberg, and Snorre. The main feature of the model is a permeability and porosity fields given on a $60 \times 220 \times 85$ Cartesian grid, in which each cell is of size $20\text{ft} \times 10\text{ft} \times 2\text{ft}$. In this specific model, the top 35 cell layers having a total height of 70 ft represent the shallow-marine Tarbert formation and the lower 50 layers having a height of 100 ft represent the fluvial Ness formation. The original geostatistical model was developed in the PUNQ project [89], and later the horizontal dimensions were scaled by a factor 1/3 to make it more heterogeneous. 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 efficient for this model [2].

The SPE 10 data set is used in a large number of publications and is publicly available from the SPE website (<http://www.spe.org/web/csp/>).

MRST supplies a module called `spe10` that downloads, reorganizes, and stores the data set in a file for later use. The module also contains routines that extract (subsets of) the petrophysical data and set up simulation models and appropriate data structures representing grids and wells. Alternatively, the data set can be downloaded using `mrstDatasetGUI`.

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 = getSPE10rock(); 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 2.12 shows porosity and permeability; the permeability tensor is diagonal with equal permeability in the two horizontal coordinate directions. 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; in other words, a deposit that has taken place close to the coastline, see Figure 2.1. 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 continental environment (see Figures 2.1 and 2.2), 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 2.13 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 bi-modal 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 trimodal.

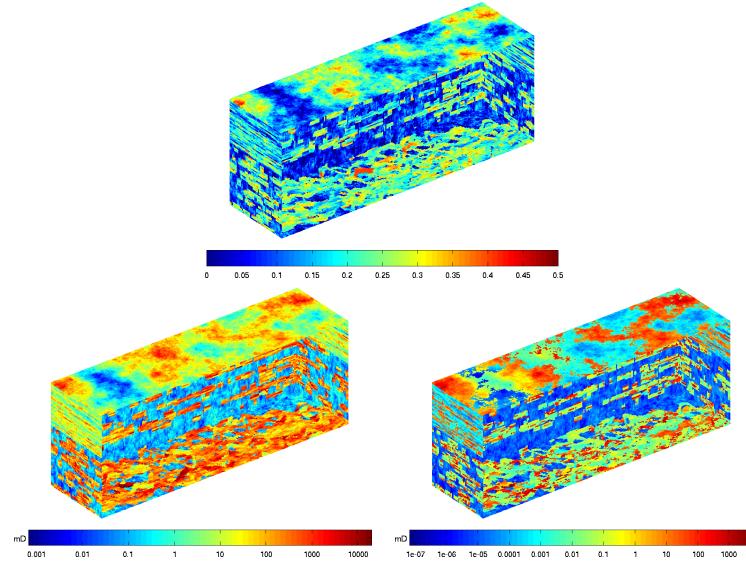


Fig. 2.12. Rock properties for the SPE 10 model. The upper plot shows the porosity, the lower left the horizontal permeability, and the lower right the vertical permeability. (The permeabilities are shown using a logarithmic color scale).

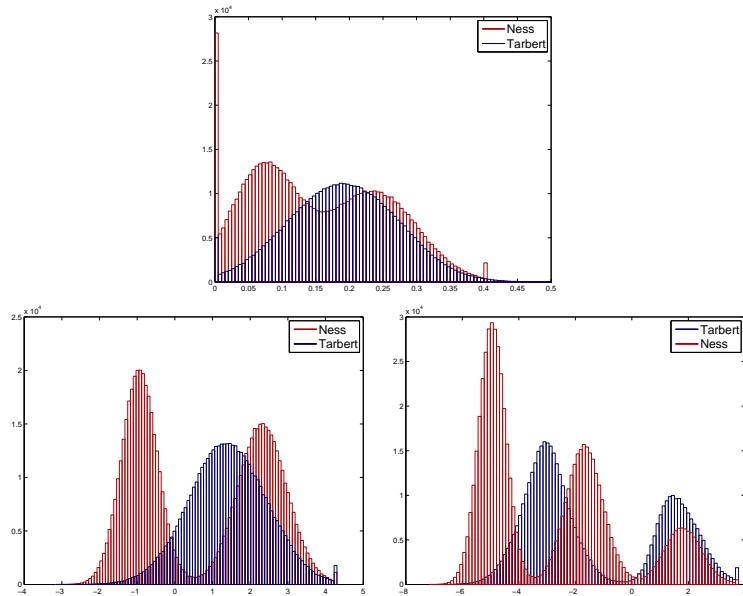


Fig. 2.13. Histogram of rock properties for the SPE 10 model: ϕ (upper plot), $\log K_x$ (lower left), and $\log K_z$ (lower right). The Tarbert formation is shown in blue and the Ness formation in red.

2.5.4 The Johansen Formation

The Johansen formation is located in the deeper part of the Sognefjord delta, 40–90 km offshore Mongstad on the west coast of Norway. A few years ago, a gas-power plant with carbon capture and storage was planned at Mongstad and the water-bearing Johansen formation was a possible candidate for storing the captured CO₂. 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. The aquifer has good sand quality and lies at a depth where CO₂ would undoubtedly be in supercritical phase, and would thus be ideal for carbon storage. With average porosities of approximately 25 percent, this implies that the theoretical storage capacity of the Johansen formation is more than one gigatonne of CO₂ [83]. 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. A set of geological models of Johansen is publicly available from the url:

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

and can be downloaded using the `mrstDatasetGUI` function. 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 ($100 \times 100 \times 11$). Herein, we consider the latter. 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 [83, 82, 11] 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 2.14 shows the porosity field of the model. The left plot shows the Dunlin shale, the Johansen sand, and the Amundsen shale, where the Johansen 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 2.15 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.

2.5.5 SAIGUP: shallow-marine reservoirs

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 top-side 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

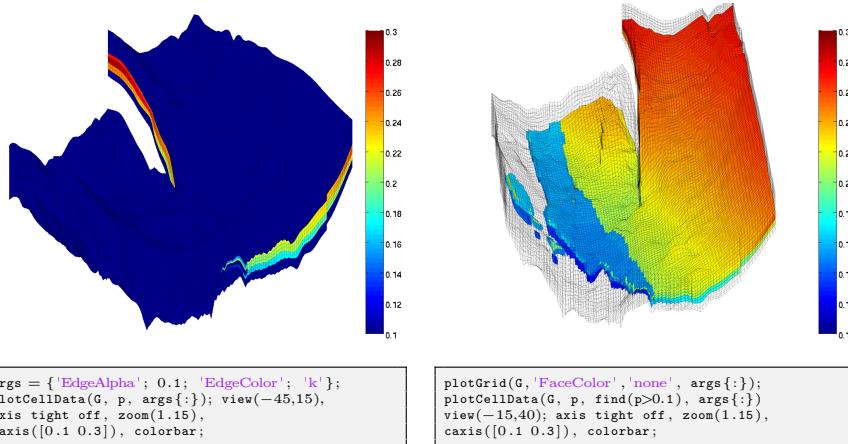


Fig. 2.14. Porosity for the Johansen data set 'NPD5'. The left plot shows porosity for the whole model, whereas in the right plot we have masked the low-porosity cells in the Amundsen and Dunlin formations.

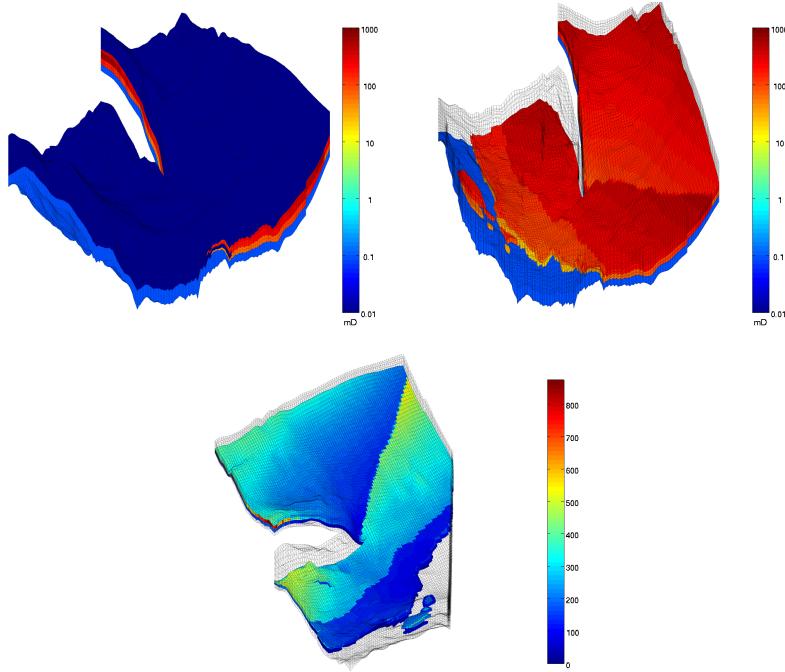


Fig. 2.15. Permeability for the Johansen data set 'NPD5'. The upper-left plot shows the permeability for the whole model, the upper-right plot shows the Johansen sand and the Amundsen shale, whereas the lower plot only shows the permeability of the Johansen sand.

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, which 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 book.

In the following, we will instead briefly outline some of the most common keywords 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 our primary example of a realistic petroleum reservoir, we will use a model from the SAIGUP study [168], whose purpose was to conduct a sensitiv-

ity analysis of the impact of geological uncertainties on production forecasting in clastic hydrocarbon reservoirs. As part of this study, a broad suite of geo-statistical realizations and structural models were generated to represent a wide span of shallow-marine sedimentological reservoirs. 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. All models are synthetic, but contain representative examples of the complexities seen in real-life reservoirs.

One of the many SAIGUP realizations is publicly available from the MRST website. 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 in the ECLIPSE format. The data set can be downloaded using the `mrstDatasetGUI` function. Here, however, we unpack the data set manually for completeness of presentation. Assuming that the archive file `SAIGUP.tar.gz` that contains the model realization has been downloaded as described on the webpage, we 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 containing 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.PERMX` 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 be used to associate different rock-fluid properties.

Recall that MRST uses the strict SI conventions in all of its internal calculations. The SAIGUP model, however, is provided using the ECLIPSE 'METRIC' conventions (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 99. The third statement constructs a rock object containing porosity, permeability, and net-to-gross.

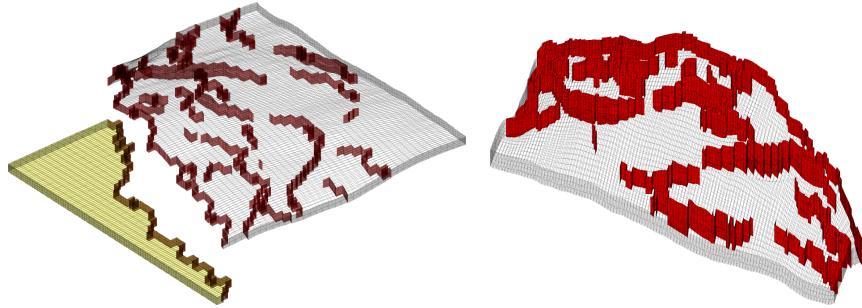


Fig. 2.16. The structural SAIGUP model. The left plot shows the full model with faults marked in red and inactive cells marked in yellow, whereas the right plot shows only the active parts of the model seen from the opposite direction.

For completeness, we first show a bit more details of the structural model in Figure 2.16. The left plot shows the whole $40 \times 120 \times 20$ grid model¹, 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 2.16 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.

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 2.17, 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 2.17.

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 2.18 along with the horizontal and vertical permeability. The plotting routines are exactly the same as for the porosity in Figure 2.17, 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

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.

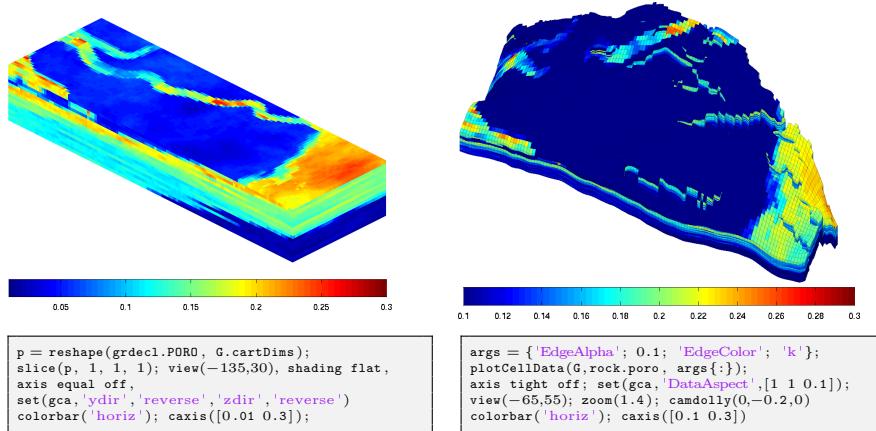


Fig. 2.17. Porosity for the SAIGUP model. The left plot shows porosity as generated by geostatistics in logical ijk space. The right plot shows the porosity mapped to the structural model shown in Figure 2.16.

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 modelled through a multiplier value (`MULTZ`) associated with each cell, which takes values between zero and one and 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 completeness, 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 identical). In Figure 2.19, 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 2.20. For completeness, the figure also shows the permeability distribution inside

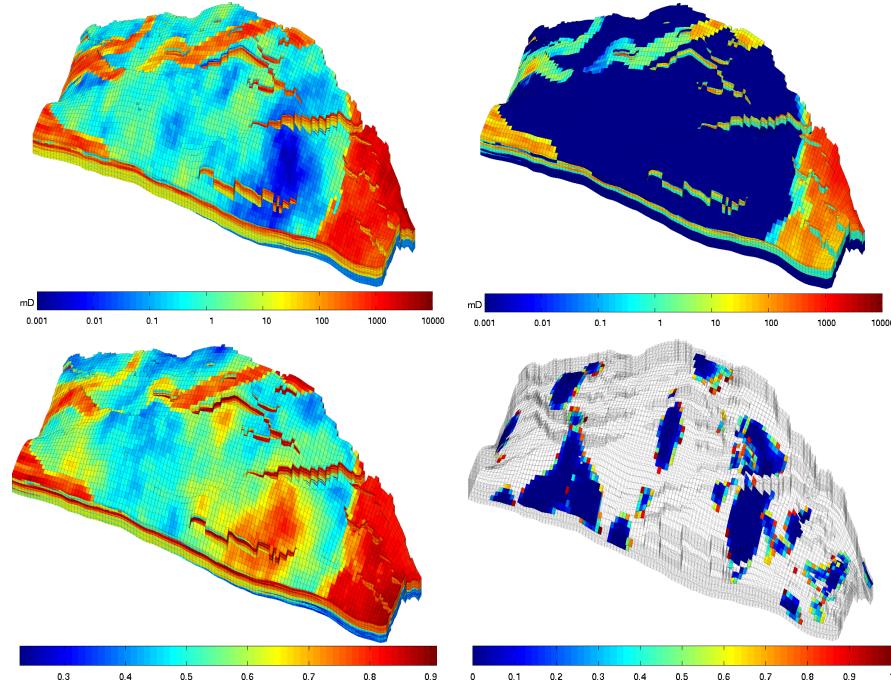


Fig. 2.18. The upper plots show the permeability for the SAIGUP model, using a logarithmic color scale, with horizontal permeability to the left and vertical permeability to the right. The lower-left plot shows net-to-gross, i.e., the fraction of reservoir rock per bulk volume. The lower-right plot shows regions of the reservoir where reduced vertical communication is modelled by vertical multiplier values less than unity.

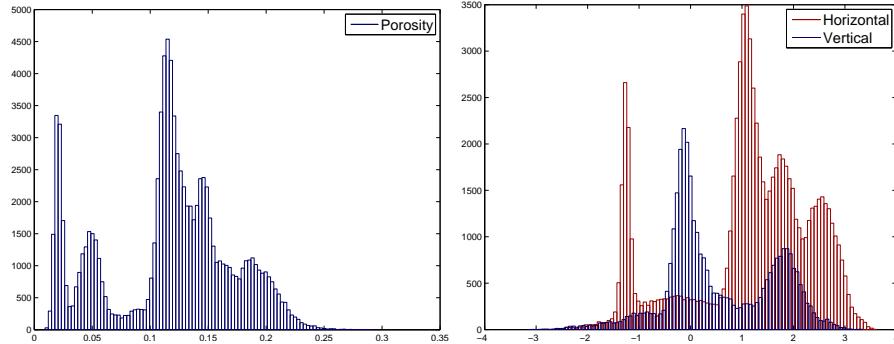


Fig. 2.19. Histogram of the porosity (left) and the logarithm of the horizontal and vertical permeability (right) for the shallow-marine SAIGUP model. Since the reservoir contains five different facies, the histograms are multi-modal. See also Figure 2.20.

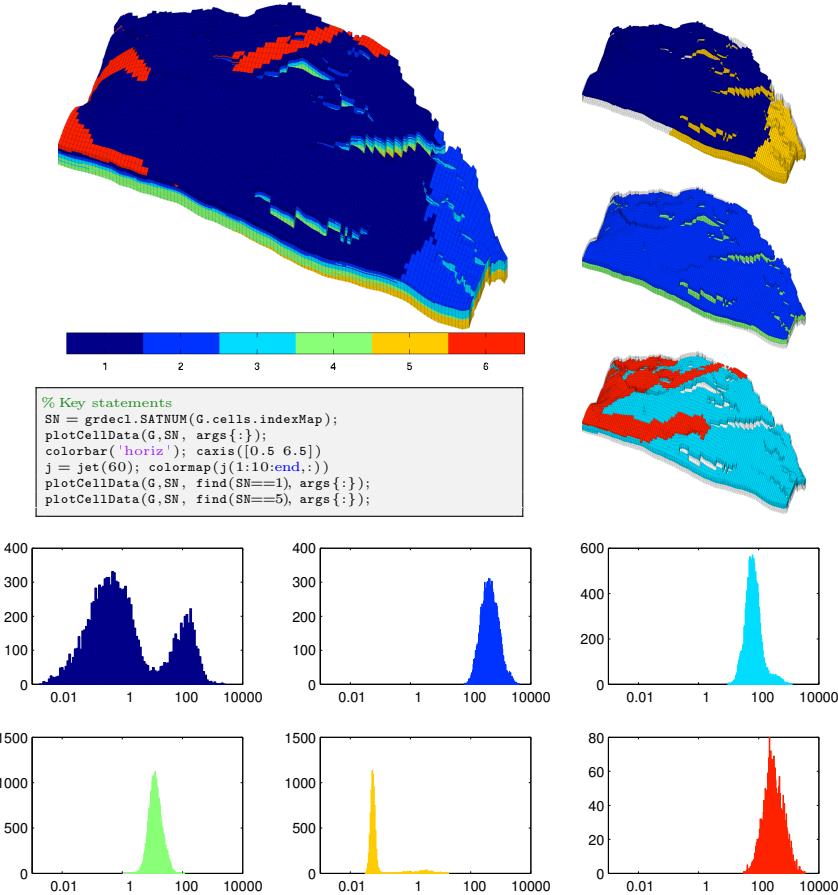


Fig. 2.20. The upper-left plot shows the rock type distribution for the SAIGUP model. The right column shows the six rock types grouped in pairs; from top to bottom, rock types number 1 and 5, 2 and 4, and 3 and 6. The bottom part of the figure shows histograms of the lateral permeability in units [mD] for each of the six rock types found in the SAIGUP model.

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 2.21 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

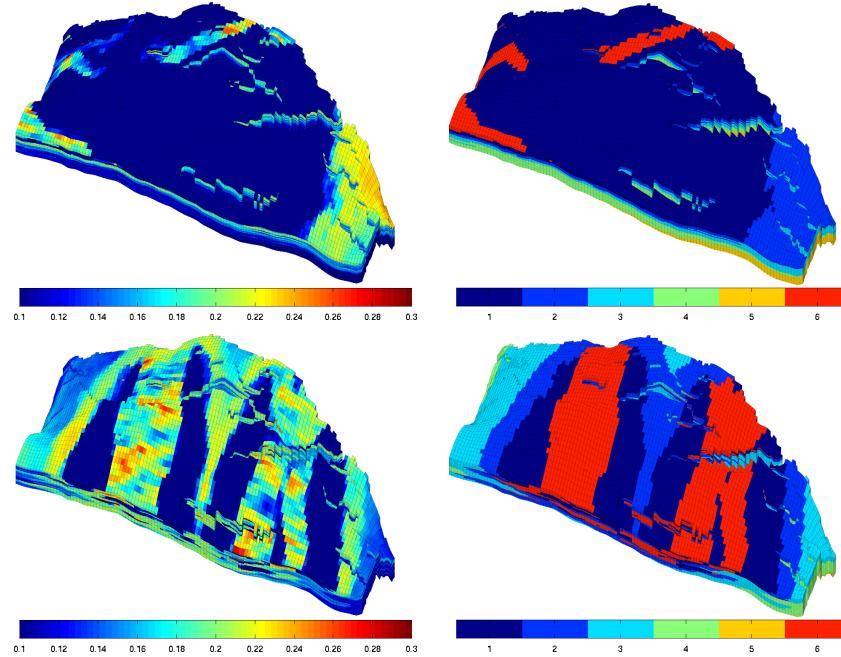


Fig. 2.21. Comparison of porosity (left) and the distribution of rock types (right) for two different SAIGUP realizations.

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.

COMPUTER EXERCISES:

1. Look at the correlation between the porosity and the permeability for the SPE 10 data set. Do you see any artifacts, and if so, how would you explain them? (Hint: plot ϕ versus $\log K$)
2. Download the CaseB4 models that represent a simple box geometry with intersecting faults. Pick at least one of the model realizations and try to set homogeneous and random petrophysical data as discussed in Sections 2.5.1 and 2.5.2.

3. The $58 \times 48 \times 10$ permeability field given in `rock1.mat` in the book module contains an unusual geological structure. Can you find what it is? (Hint: if you want to explore the model interactively, you can try to use `plotToolbar` from the `mrst-gui` module).
4. Download the `BedModels1` and `BedModel2` data sets that represent sedimentary beds similar to the facies model shown in Figure 2.7. Use the techniques introduced in Sections 2.5.3 to 2.5.5 to familiarize yourself with these models:
 - look at porosities and permeabilities in physical space
 - compare with the same quantities in ijk space
 - find models that have facies information and look at the distribution of petrophysical properties inside each facies
5. Modify the `simpleGravityColumn` example from Section 1.4 so that it uses the geometry and petrophysical data in the `mortarTestModel` or `periodicTilted` models from the `BedModels1` data set instead. Can you explain what you observe?

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

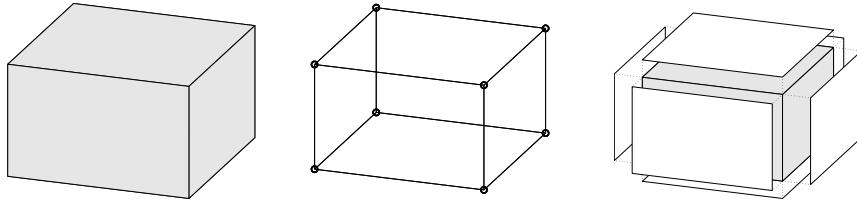


Fig. 3.1. Illustration of a single cell (left), vertices and edges (middle), and cell faces (right).

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, and so on lead to cells that can have quite ir-

regular 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.

A number of videos that complement the material presented in this chapter can be found in the second MRST Jolt [155]. This Jolt introduces different types of grids, discusses how such grids can be represented, and outlines functionality in MRST you can use to generate your own grids.

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 $n_2 \times n_2 \times \dots \times n_k$ of cells that cover a bounded domain $[0, L_1] \times [0, L_2] \times \dots \times [0, L_k]$.

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.

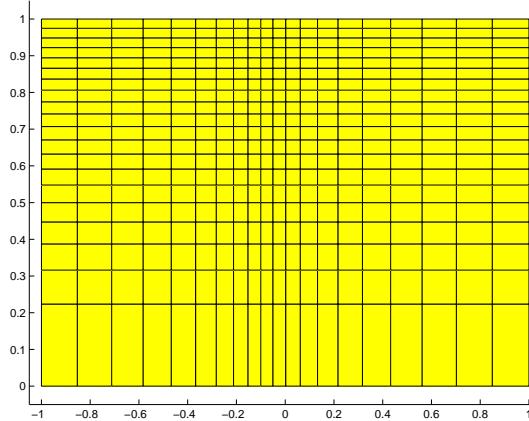


Fig. 3.2. Example of a rectilinear grid.

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 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, you can create such grids by first instantiating a regular Cartesian or a rectilinear grid and then manipulating the vertices. 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 [226, 29] 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);

```

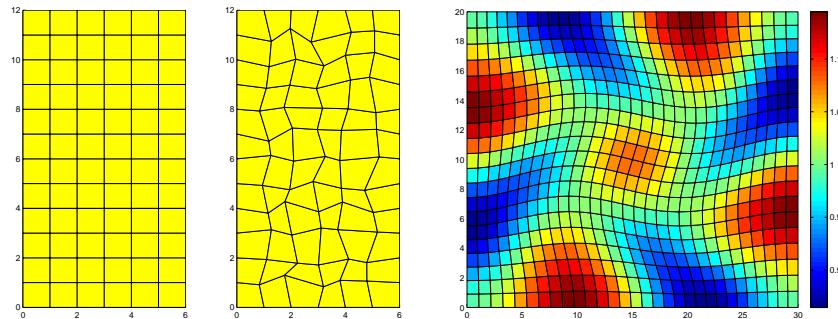


Fig. 3.3. The middle plot shows a rough grid created by perturbing all internal nodes of the regular 6×12 Cartesian grid in the left plot. The right plot shows a curvilinear grid created using the function `twister` that uses a combination of sin functions to perturb a rectilinear grid. The color is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

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

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 2.16) 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;
```

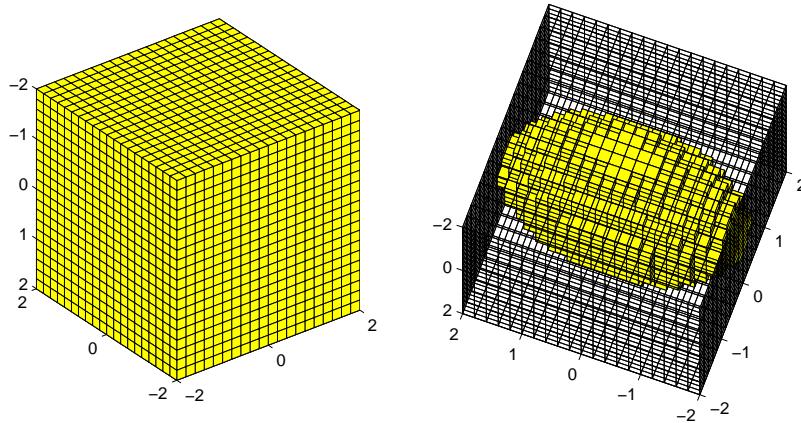


Fig. 3.4. Example of a regular Cartesian grid representing a domain in the form of an ellipsoid. The underlying logical Cartesian grid is shown in the left plot and as a wireframe in the right plot. The active part of the model is shown in yellow color in the right plot.

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

You can find more examples of how you can make structured grids and populate them with petrophysical properties in the fourth video of the second MRST Jolt [155].

COMPUTER EXERCISES:

6. Make the grid shown below:

Hint: the grid spacing in the x -direction is given by $\Delta x(1 - \frac{1}{2} \cos(\pi x))$ and the colors signify cell volumes.

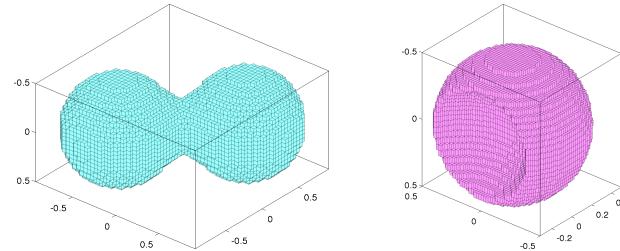
7. Metaballs are commonly used in computer graphics to generate organic-looking objects. Each metaball is defined as a smooth function that has finite support. One example is

$$m(\vec{x}, r) = \left[1 - \min\left(\frac{|\vec{x}|^2}{r^2}, 1\right) \right]^4.$$

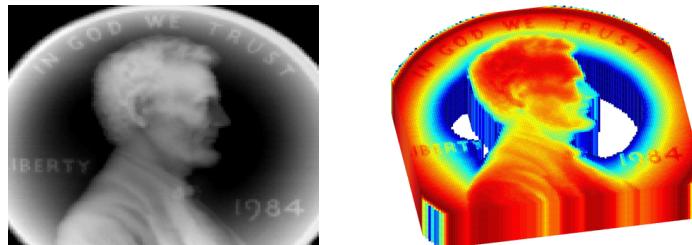
Metaballs can be used to define objects implicitly, e.g., as all the points \vec{x} that satisfy

$$\sum_i m(\vec{x} - \vec{x}_i, r_i) \leq C, \quad C \in \mathbb{R}^+$$

Use this approach and try to make grids similar to the ones shown below:



8. A simple way to make test models with funny geometries is to use the method of fictitious domains and let an image define the domain of interest. In the example below, the image was taken from `penny`, which is one of the standard data sets that are distributed with MATLAB, and then used to define the geometry of the grid and assign permeability values



Pick your own favorite image or make one in a drawing program and use `imread` to load the image into MATLAB as a 3D array, which you can use to define your geometry and petrophysical values. If you do not have an image at hand, you can use `penny` or `spine`. For `penny`, in particular, you may have to experiment a bit with the threshold used to define your domain to ensure that all cells are connected, i.e., that the grid you obtain consists of only one piece.

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} ,

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

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.

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

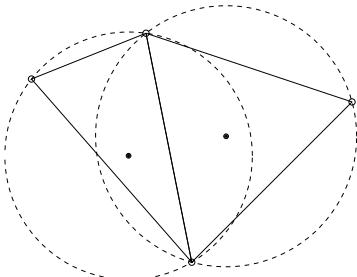


Fig. 3.5. Two triangles and their circumcircles.

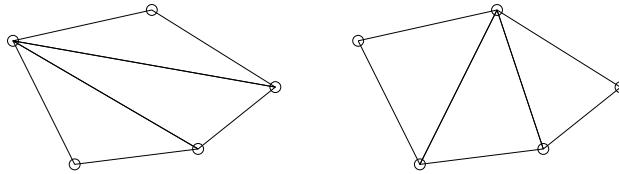


Fig. 3.6. Example of two triangulations of the same five points; the triangulation to the right satisfies the min-max criterion.

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.

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 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 built-in `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

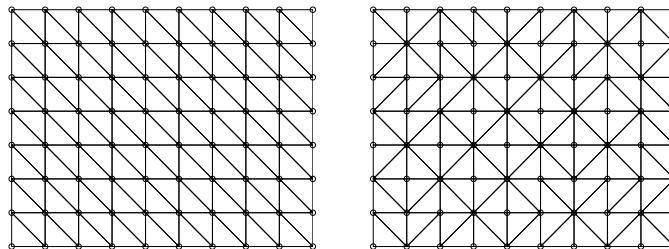


Fig. 3.7. Two different Delaunay tessellations of a rectangular point mesh.

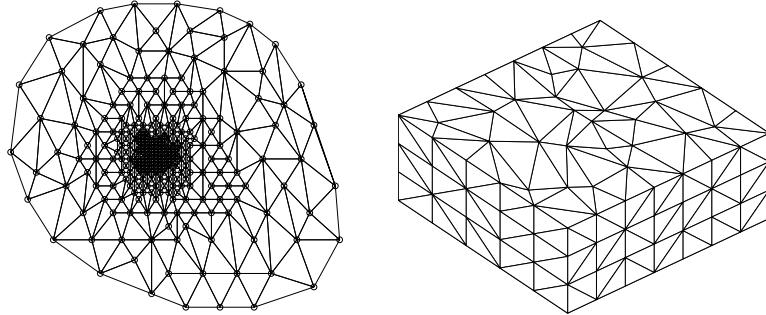


Fig. 3.8. The left plot shows the triangular grid from the `seamount` demo case. The right plot shows a tetrahedral tessellation of a 3D point mesh.

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 polytopes¹ 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

$$V(x_i) = \left\{ x \mid \|x - x_i\| < \|x - x_j\| \forall j \neq i \right\}. \quad (3.1)$$

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).

¹ A polytope is a generic term that refers to a polygon in 2D, a polyhedron in 3D, and so on.

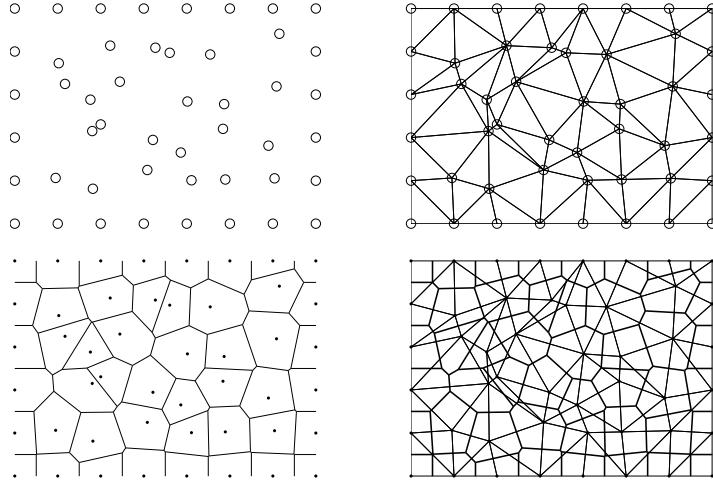


Fig. 3.9. Duality between Voronoi diagrams and Delaunay triangulation. From top left to bottom right: generating points, Delaunay triangulation, Voronoi diagram, and Voronoi diagram (thick lines) and Delaunay triangulation (thin lines).

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 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 does not

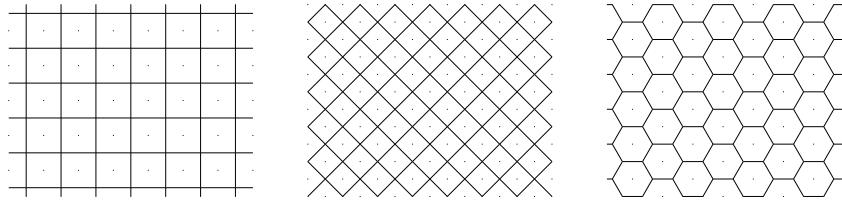


Fig. 3.10. Three examples of Voronoi diagrams generated from 2D point lattices. From left to right: square lattice, square lattice rotated 45 degrees, lattice forming equilateral triangles.

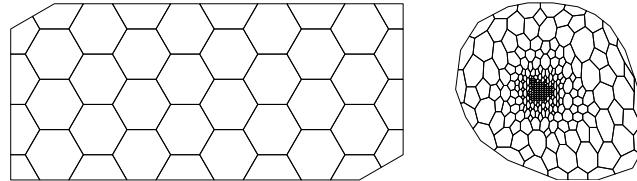


Fig. 3.11. Two examples of Voronoi grids. The left plot shows a honeycombed PEBI grid and the right plot shows the PEBI grid derived from the `seamount` demo case.

yet have a similar function that generates a Voronoi grid from a point set, but offers `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 generate a honeycombed grid similar to the one shown in the right plot in Figure 3.10

```
[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

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

Several of the examples discussed above can also be found in the fifth video of the second MRST Jolt [155]. Since `pebi` is a 2D code, we cannot apply it directly to the 3D tetrahedral grid shown in Figure 3.8 to generate a dual 3D Voronoi grid. In the next section, we discuss a simple approach in geological modeling in which we extrude 2D Voronoi grid to 3D to preserve geological layering. The interested reader should consult [174] and references therein for more discussion of general 3D Voronoi grids.

3.2.3 Other types of tessellations

Tessellations come in many other forms than the Delaunay and Voronoi types discussed in the two previous sections. Tessellations are more commonly referred to as tilings and are patterns made up of geometric forms (tiles) that are repeated over and over without overlapping or leaving any gaps. Such tilings can be found in many patterns of nature, like in honeycombs, giraffe skin, pineapples, snake skin, tortoise shells, to name a few. Tessellations have also been extensively used for artistic purposes since ancient times, from the decorative tiles of Ancient Rome and Islamic art to the amazing artwork of M. C. Escher. For completeness (and fun), MRST offers the function `tessellationGrid` that can take a tessellation consisting of symmetric n -polygons and turn it into a correct grid structure. While this may not be very useful in modeling petroleum reservoirs, it can easily be used to generate irregular grids that can be used to stress-test various discretization methods. Let us first use it to make a standard $n \times m$ Cartesian mesh:

```
[x,y] = meshgrid(linspace(0,1,n+1),linspace(0,1,m+1));
I = reshape(1:(n+1)*(m+1),m+1,n+1);
T = [reshape(I(1:end-1,1:end-1),[],1)'; reshape(I(1:end-1,2:end ),[],1)';
      reshape(I(2:end , 2:end ),[],1)'; reshape(I(2:end , 1:end-1),[],1)'];
G = tessellationGrid([x(:) y(:)], T);
```

Here, the vertices and cells are numbered first in the y direction and then in the x -direction, so that the first two lines in `T` read `[1 m+2 m+3 2; 2 m+3 m+4 3]`, and so on.

There are obviously many ways to make more general tilings. The script `showTessellation` in the `book` module shows two slightly different approaches. To generate the alternating convex/concave hexagonal tiling illustrated in Figure 3.12 we first generate the convex and the concave tiles. These will have symmetry lines that together form a triangle for each tile, which we can use to glue the tiles together. If we glue a convex (blue) to the right edge of the concave (yellow) tile and likewise glue a concave (yellow) tile to the right of the upper convex (blue) tile, we get a dodecagon consisting of two convex and two concave tiles. This composite tile can now be placed on a regular mesh; in Figure 3.12 we have used a 4×2 regular mesh.

To generate the tiling shown in Figure 3.13 we start from an equilateral triangulation (p, t) covering a certain part of space. We then extract the endpoints p_1 and p_2 on each edge and compute the angle ϕ the line between them makes with the x -axis. We will use this information to perturb the points. By using this orientation of the lines, we can easily make sure that the original triangles can be turned into matching $3n$ -polygons for $n = 2, 3, \dots$, if we for each triple of new points we add, describe the point added to the original $p_1 p_2$ line on the form (with α and β constants for each triplet):

$$x = p_1 + \alpha|p_2 - p_1|[\cos(\phi + \beta), \sin(\phi + \beta)].$$

Figure 3.13 shows that consecutive addition of four points on each original line segment, thereby turning a triangulation into a pentadecagonal tessellation.

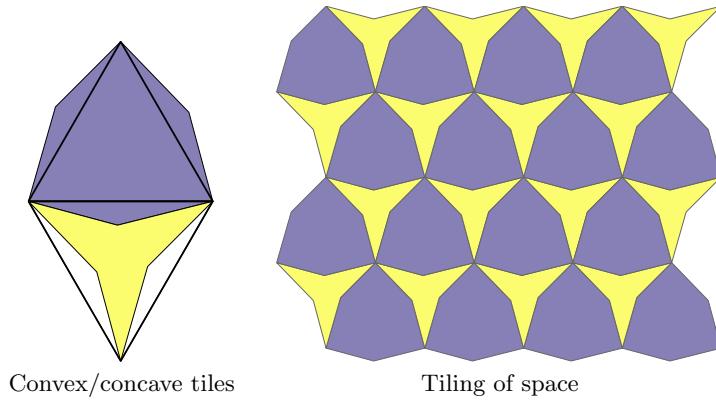


Fig. 3.12. Tiling consisting of alternating convex and concave hexahedrons.

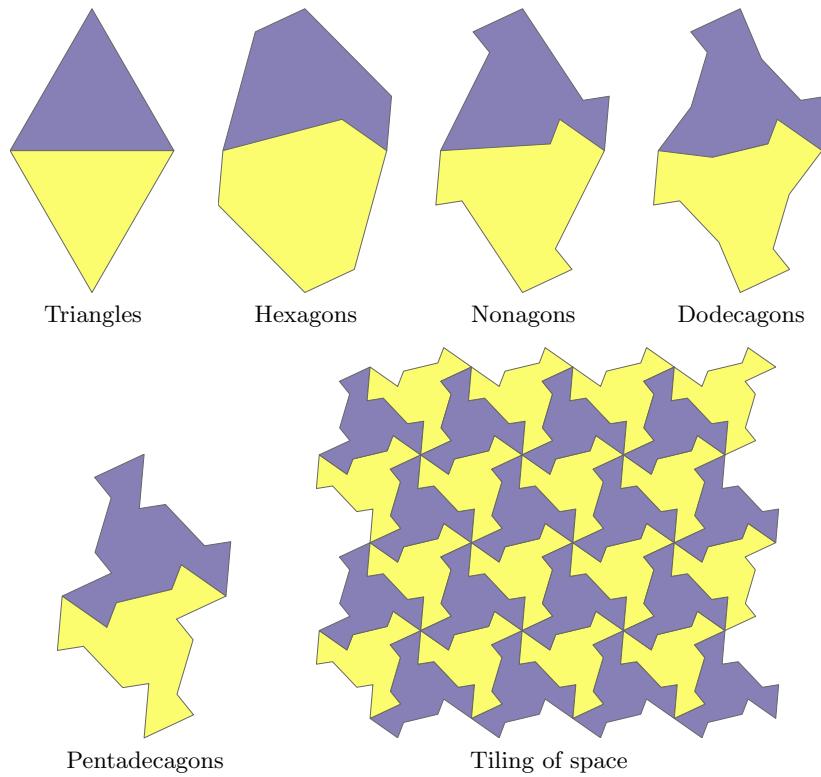


Fig. 3.13. Gradual creation of a tiling consisting of irregular pentadecagons. (Can you see the hen facing left and the man running towards the right?).

3.2.4 Using an external mesh generator

Using a Delaunay triangulation as discussed above, we can generate grids that fit a given set of vertices. However, in most applications, the vertex points are not given *a priori* and all one wants is a reasonable grid that fits an exterior boundary describing the perimeter of the domain and possibly also a set of interior boundaries and/or features, which in the case of subsurface media could be faults or major fractures. To this end, one typically will have to use a mesh generator. There are a large number of mesh generators available online, and in principle any of these can be used in combination with MRST's grid factory routines as long as they produce triangulations on the form outlined above. My personal favorite is `DistMesh` by Persson and Strang [212]. While most mesh generators tend to be complex and quite inaccessible codes, `DistMesh` is a relatively short and simple MATLAB code written in the same spirit as MRST. The performance of the code may not be optimal, but the user can go in and inspect all algorithms and modify them to his or her purpose. In the following, we will use `DistMesh` to generate a few examples of more complex triangular and Voronoi grids in 2D.

`DistMesh` is distributed under the GNU GPL license (which is the same license that MRST uses) and can be downloaded from the software's webpage. The simplest way to integrate `DistMesh` with MRST is to install it as a 3rd-party module. Assuming that you are connected to internet, this is done as follows:

```
path = fullfile(ROOTDIR, 'utils', '3rdparty', 'distmesh');
mkdir(path)
unzip('http://persson.berkeley.edu/distmesh/distmesh.zip', path);
mrstPath('reregister', 'distmesh', path);
```

You are now ready to start using the software. If you intend to use `DistMesh` many times, you should copy the last line to the `startup_user.m` file in the MRST root directory.

In `DistMesh`, the perimeter of the domain is represented using a signed distance function $d(x, y)$, which is by definition set to be negative inside the region. The software offers a number of utility functions that makes it simple to describe relatively complex geometries, as we shall see in the following. Let us start with a simple example, which is taken from the `DistMesh` webpage: Consider a square domain $[-1, 1] \times [-1, 1]$ with a circular cutout of radius 0.5 centered at the origin. We start by making a grid that has a uniform target size $h = 0.2$

```
mrstModule add distmesh;
fd=@(p) ddifff(drectangle(p,-1,1,-1,1), dcircle(p,0,0,0.5));
[p,t]=distmesh2d(fd, @uniform, 0.2, [-1,-1;1,1], [-1,-1;-1,1;1,-1;1,1]);
G = triangleGrid(p, t);
```

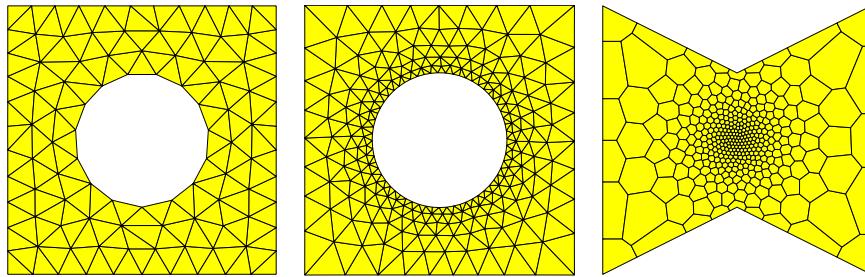


Fig. 3.14. Grids generated by the `DistMesh` grid generator.

Here, we have used utility functions `ddif`, `drectangle` and `dcircle` to compute the signed distance from the outer and the inner perimeter. Likewise, the `huniform` to set enforce uniform cell size equal distance between the points in the initial, which here is 0.2. The fourth argument is the bounding box of our domain, and the fifth argument consist of fixed points that the algorithm is not allowed to move. After the triangulation has been computed by `distmesh2d` we pass it to `triangleGrid` to make a MRST grid structure. The resulting grid is shown to the left in Figure 3.14.

As a second test, let us make a graded grid that has a mesh size of approximately 0.05 at the inner circle and 0.2–0.35 at the outer perimeter. To enforce this, we replace the `huniform` function by another function that gives the correct mesh size distribution

```
fh=@(p) 0.05+0.3*dcircle(p,0,0,0.5);
[p,t]=distmesh2d(fd, fh, 0.05, [-1,-1;1,1], [-1,-1;-1,1;1,-1;1,1]);
```

The resulting grid is shown in the middle plot in Figure 3.14 and has the expected grading from the inner boundary and outwards to the perimeter.

In our last example, we will create a graded triangulation of a polygonal domain and then use `pebi` to compute its Voronoi diagram

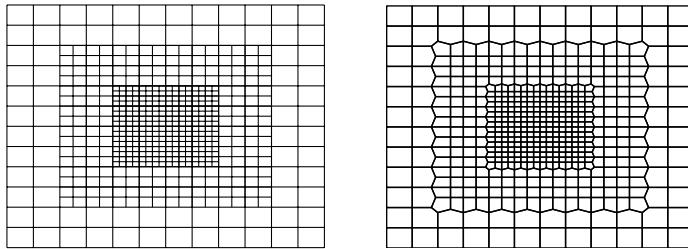
```
pv = [-1 -1; 0 -.5; 1 -1; 1 1; 0 .5; -1 1; -1 -1];
fh = @(p,x) 0.025 + 0.375*sum(p.^2,2);
[p,t] = distmesh2d(@dpoly, fh, 0.025, [-1 -1; 1 1], pv, pv);
G = pebi(triangleGrid(p, t));
```

Here, we use the utility function `dpoly(p,pv)` to compute the signed distance of any point set `p` to the polygon with vertices in `pv`. Notice that `pv` must form a closed path. Likewise, since the signed distance function and the grid density function are assumed to take the same number of arguments, `fh` is created with a dummy argument `x`. The argument sent to these functions are passed as the sixth argument to `distmesh2d`. The resulting grid is shown to the right in Figure 3.14.

`DistMesh` also has routines for creating n D triangulations and triangulations of surfaces, but these are beyond the scope of the current presentation.

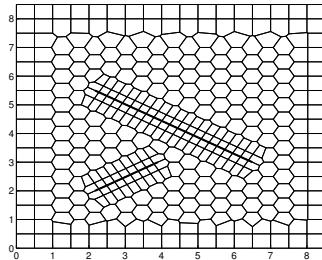
COMPUTER EXERCISES:

9. Create MRST grids from the standard data sets `trimesh2d` and `tetmesh`. How would you assign lognormal petrophysical parameters to these grids so that the spatial correlation is preserved?
10. In MRST all triangular grids are assumed to be planar so that each vertex can be given by a 2D coordinate. However, triangular grids are commonly used to represent non-planar surfaces in 3D. Can you extend the function `triangleGrid` so that it can construct both 2D and 3D grids? You can use the data set `trimesh3d` as an example of a triangulated 3D surface.
11. MRST does not yet have a grid factory routine to generate structured grids with local, nested refinement as shown in the figure to the left below.



Try to use a combination of `triangleGrid` and `pebi` to make a good approximation to such a grid as shown to the right in the figure above. (Hint: to get rid of artifacts, one layer of cells were removed along the outer boundary.)

12. The figure below shows an unstructured hexagonal grid that has been adapted to two faults in the interior of the domain and padded with rectangular cells near the boundary. Try to implement a routine that generates a similar grid.



Hint: in this case, the strike direction of the faults are $\pm 30^\circ$ and the start and endpoints of the faults have been adjusted so that they coincide with the generating points of hexagonal cells.

13. What would you do to fit the tessellations in Figures 3.12 and 3.13 so that they fill a rectangular box without leaving any gaps along the border?
14. Download and install `distmesh` and try to make MRST grids from all the triangulations shown in [212, Fig. 5.1].

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 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 occurred 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. A complementary discussion is given in videos 2, 6, and 7 of the second MRST Jolt [155].

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 [217], which we already encountered in Chapter 2.5. 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

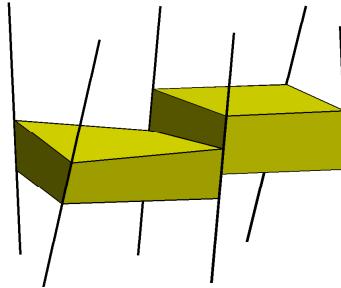


Fig. 3.15. Each cell in the corner-point grid is restricted by four pillars and two points on each pillar.

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 3.15. 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 2.16, are present in the topological ijk -numbering but are indicated by a zero porosity or net-to-gross value, as discussed in Chapter 2.4 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 3.16. 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 3.17. In a similar manner, (simple) vertical and inclined faults can be easily modelled by aligning the pillars with fault 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.

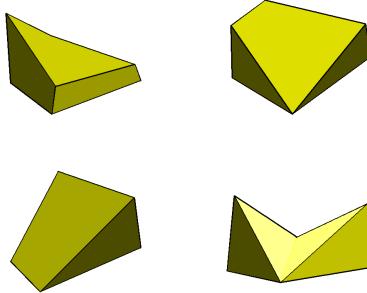


Fig. 3.16. Examples of deformed and degenerate hexahedral cells arising in corner-point grid models.

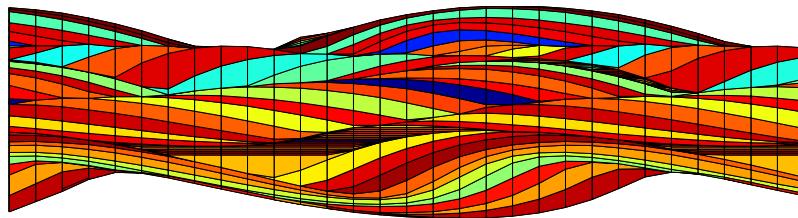


Fig. 3.17. Side view in the xx -plane of corner-point grid with vertical pillars modeling a stack of sedimentary beds (each layer indicated by a different color).

To illustrate the concepts introduced so far, we consider a low-resolution version of the model from Figure 2.11 on page 46 created by the `simpleGrdecl` grid-factory routine, which generates an input stream containing the basic keywords that describe a corner-point grid in the ECLIPSE input deck [226, 29]

```
grdecl = simpleGrdecl([4, 2, 3], .12, 'flat', true);
```

```
grdecl =
    cartDims: [4 2 3]
    COORD: [90x1 double]
    ZCORN: [192x1 double]
    ACTNUM: [24x1 int32]
```

The 5×3 mesh of pillars are given in terms of a pair of 3D coordinates for each pillar in the `COORD` field, whereas the z -values that determine vertical positions uniquely along each pillar for the eight corner-points of the 24 cells are given in the `ZCORN` field. To extract these data, we use two MRST routines

```
[X,Y,Z] = buildCornerPtPillars(grdecl,'Scale',true);
[x,y,z] = buildCornerPtNodes(grdecl);
```

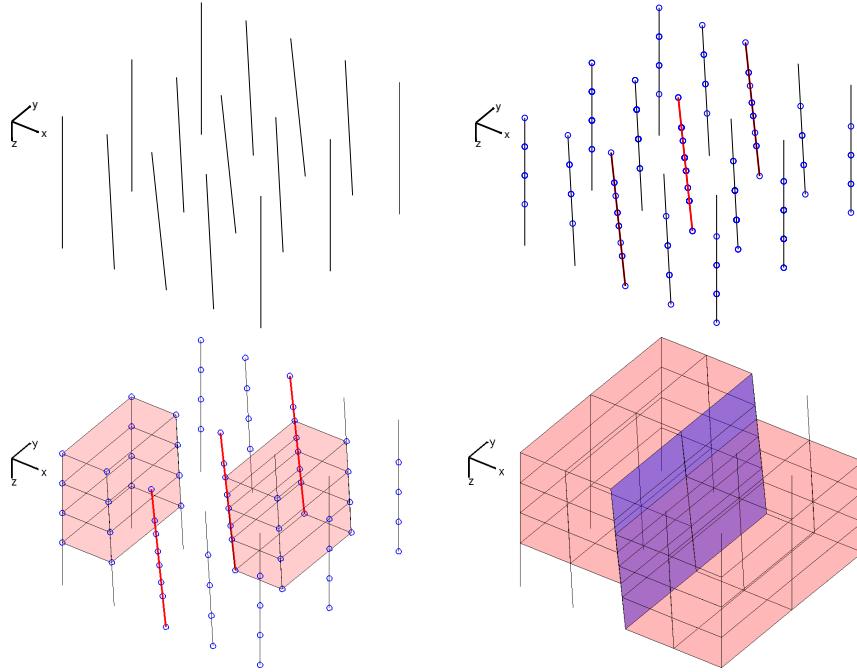


Fig. 3.18. Specification of a corner-point grid. Starting from the pillars (upper left), we add corner-points and identify pillars containing non-matching corner marked in red (upper right). A stack of cells is created for each set of four pillars (lower left), and then the full grid is obtained (lower right). In the last plot, the fault faces have been marked in blue.

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 3.18, 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:

```
% 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 3.19 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 3.19, 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 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 3.19, 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 direction. To model this scenario using corner-point grids, we basically have two 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

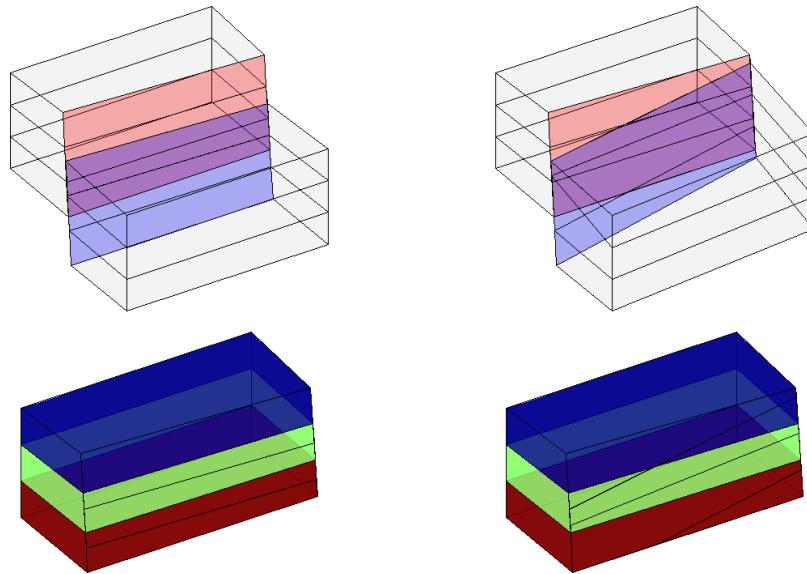


Fig. 3.19. Subdivision of fault face in two three-dimensional models. In the left column, the subfaces are all rectangular. In the right column they are not. In both the upper plots, the faces marked in red belong only to the cells behind the fault surface, the blue faces belong only to the cells in 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.

in a subsequent simulation, as will be discussed in more detail in Chapter 6. Alternatively, we can choose a vertical extrusion direction and replace deviated fault surfaces by stair-stepped approximations so that the faults zigzag in direction not aligned with the grid. This will create cells that are mostly K-orthogonal and less prone to grid-orientation errors.

Figure 3.20 shows two different grid models, taken from the CaseB4 data set. In the stair-stepped model, the use of cells with orthogonal faces causes the faults to be represented as zigzag patterns. The pillar grid correctly represents the faults as inclined planes, but has cells with degenerate geometries and cells that deviate strongly from being orthogonal in the lateral direction. Likewise, some pillars have 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 Jurassic sandstone at a depth of 2500 meter below sea level, and was originally estimated to contain 90.8 million m³ oil, mainly in the Ile and Tofte formations, and 12.0 billion m³ in the Garn formation. The field is operated by

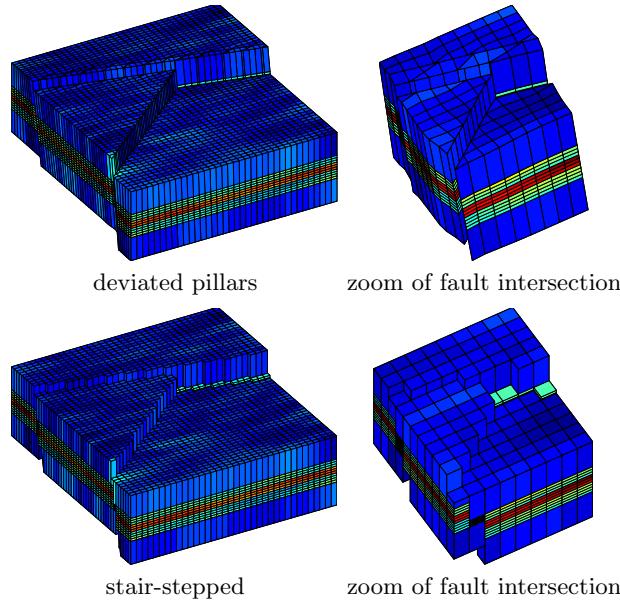


Fig. 3.20. Modeling the intersection of two deviated faults using deviated pillars (top) and stair-stepped approximation (bottom). CaseB4 grids courtesy of Statoil.

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 purposes². More recently, the Open Porous Media (OPM) initiative (opm-project.org) released the full simulation model as an open data set on Github (github.com/OPM/opm-data). The data set can either be downloaded and installed using `mrstDatasetGUI` or directly from the command line

```
makeNorneSubsetAvailable() && makeNorneGRDECL()
```

Once in place, the data set can then be loaded as follows:

```
grdecl = readGRDECL(fullfile(getDatasetPath('norne'), 'NORNE.GRDECL'));
grdecl = convertInputUnits(grdecl, getUnitSystem('METRIC'));
```

² The Norne Benchmark data sets are hosted and supported by the Center for Integrated Operations in the Petroleum Industry (IO Center) at NTNU (<http://www.ipt.ntnu.no/~norne/>). The data set used herein was first released as part of “Package 2: Full field model” (2013)

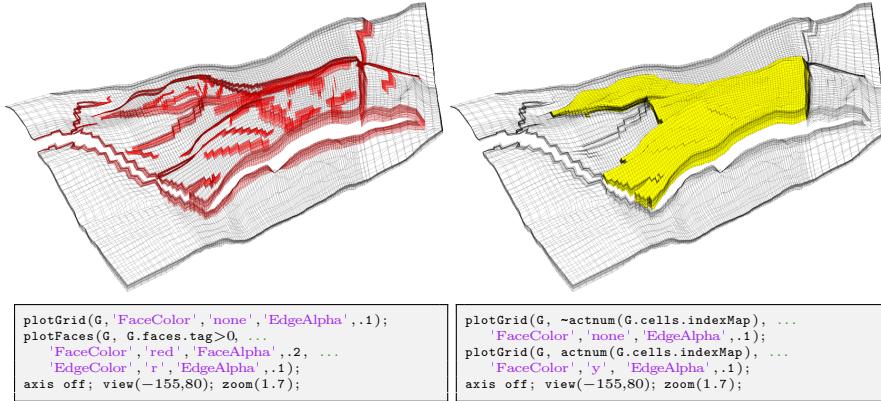


Fig. 3.21. The Norne field from the Norwegian Sea. The plots show the whole grid with fault faces marked in red (left) and active cells marked in yellow (right).

The model consists of $46 \times 112 \times 22$ corner-point cells. We start by plotting the whole model, including inactive cells. To this end, we need to override³ 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 3.21. 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 [226, 29]. In particular, we will examine some parts of the model in more detail. To this end, we will use the 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 3.22, 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

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

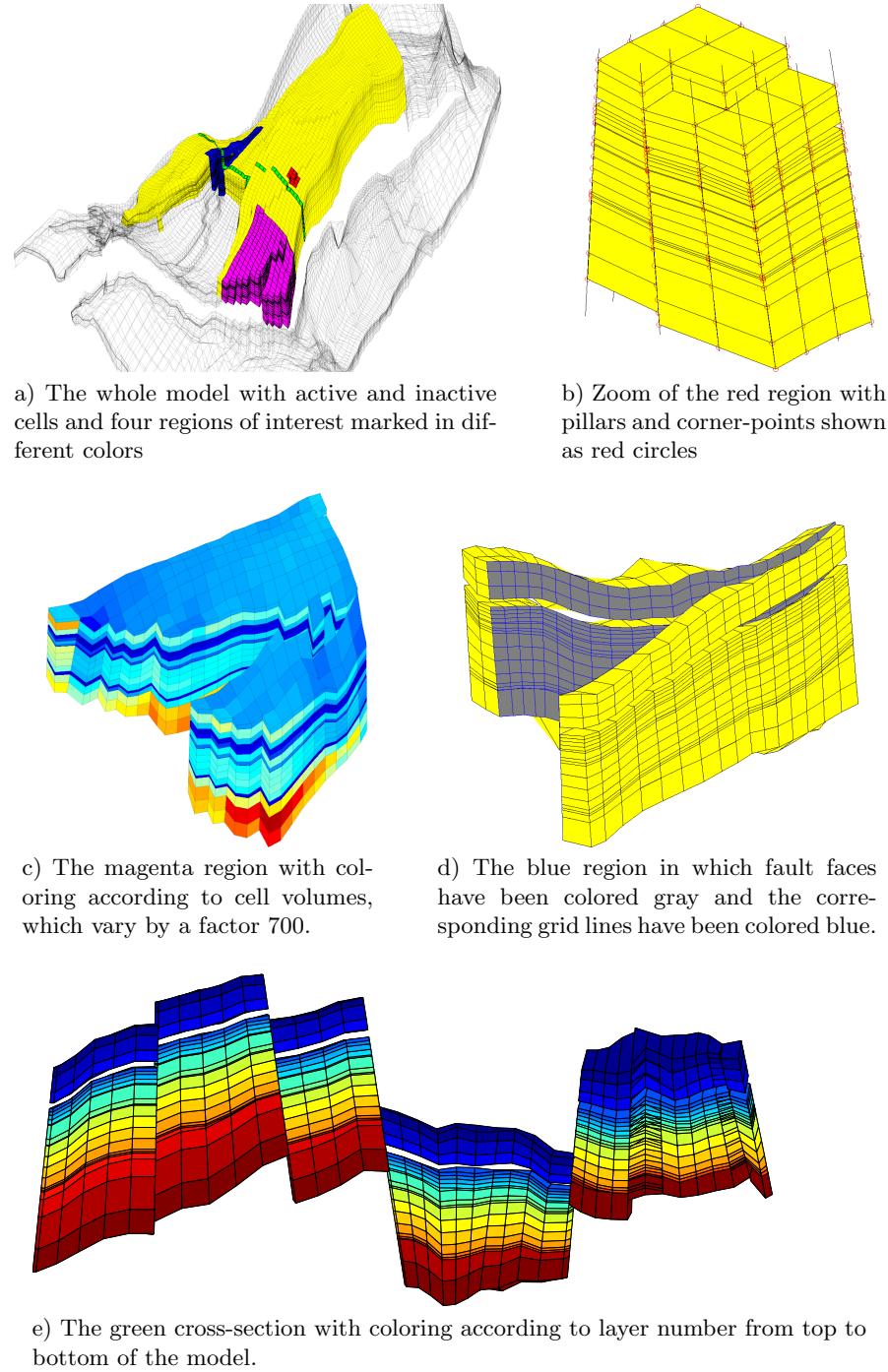


Fig. 3.22. Detailed view of subsets from the Norne simulation model.

fault surface that has been approximated by a surface that zigzags 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 thinnest 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 3.17. 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 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 will be 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.

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 3.22. 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

and produce reasonable graphical output. In general, however, we strongly advice that '`checkgrid`' remains set in its default state of `true`.

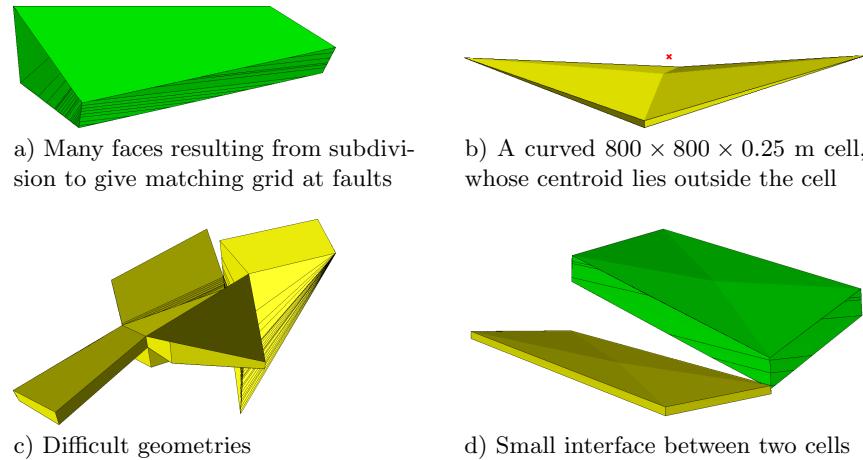


Fig. 3.23. Illustration of difficulties and challenges associated with real-life corner-point geometries.

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 areas between cell faces on different sides a fault, and so on. All these factors contribute to make fault geometries 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 3.16). 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 3.23 illustrates some geometrical and topological challenges seen in standard grid models.

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).

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

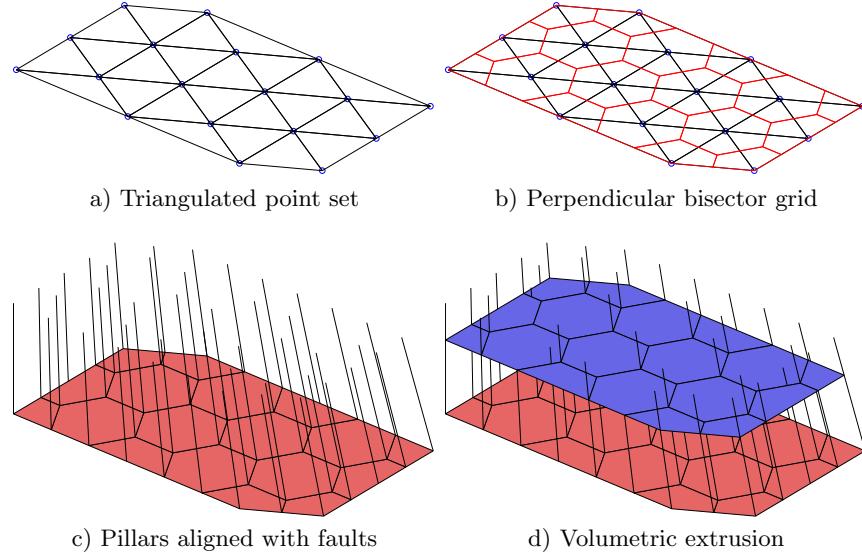


Fig. 3.24. Illustration of a typical process for generating 2.5D PEBI grids.

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.

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 2.5D unstructured 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 grids are often used to overcome the problem of areal adaptation. These grids have been designed to combine the advantages of two different gridding methods: the (areal) flexibility of unstructured grids and the simple topology of Cartesian grids in the vertical direction. The 2.5D grids are constructed in much the same way as corner-point grids, but instead of

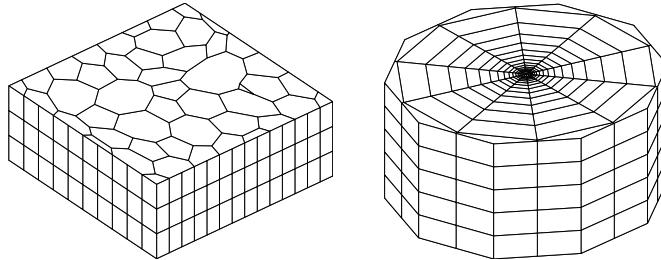


Fig. 3.25. The left plot shows a 2.5D Voronoi grid derived from a perturbed 2D point mesh extruded in the z -direction, whereas the right plot shows a radial grid.

defining pillars using a structured areal mesh, the pillars are defined based on an unstructured lateral grid. To generate such a grid, one starts by defining an areal tessellation on a surface that either aligns with the lateral direction or one of the major geological horizons. Then a pillar is introduced through each vertex in the areal grid. The pillars can either be vertical, inclined to gradually align with major fault planes as shown for the corner-point grid in Figure 3.20, or be defined so that they connect pairs of vertices in two areal tessellations placed above each other, e.g., so that these are aligned with two different geological horizons. Figure 3.24 shows the key steps in the construction of a simple 2.5D PEBI grid. Starting from a set of generating points, an areal tessellation is formed by first computing the Delaunay triangulation and then constructing a perpendicular bisector grid. Through each vertex in the areal tessellation, we define a pillar, whose angle of inclination will change from 90 degrees for vertices on the far left to 45 degrees for vertices on the far right. The pillars are then used to extrude the areal tessellation to a volumetric grid. The resulting volumetric grid 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 a quite large freedom in choosing the size and shape of the grid cells to adapt to complex features such as curved faults or to improve the areal resolution in near-well zones.

As a first example of a 2.5D grid, 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.

```
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
```

The resulting grid is shown in the left plot of Figure 3.25 and should be contrasted to the 3D tetrahedral tessellation shown to the right in Figure 3.8

As a second example, we will generate a PEBI grid with radial symmetry, which 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
```

Figure 3.25 shows the resulting grid. 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 3.26 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 3.26a in which these thin layers appear as if they were thick lines. Figure 3.26b 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 3.26e, it may appear as if the reservoir was formed by sediments being deposited along a sloping valley that ends in a flat plain. Figures 3.26c 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 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.

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

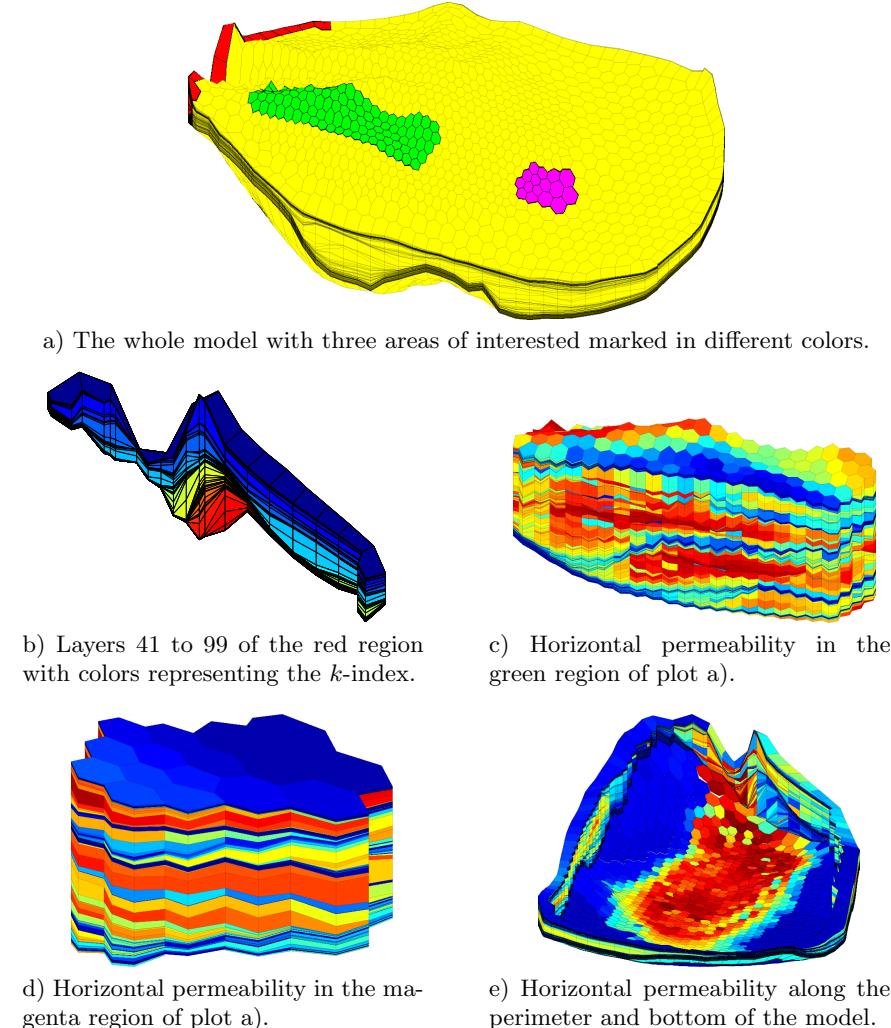


Fig. 3.26. A real petroleum reservoir modelled by a 2.5D PEBI grid having 1174 cells in the lateral direction and 150 cells along each pillar. Only 90644 out of the 176100 cells are active. The plots show the whole model as well as selected details.

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
```

This will bring you an overview of all the grid structure and all its members.

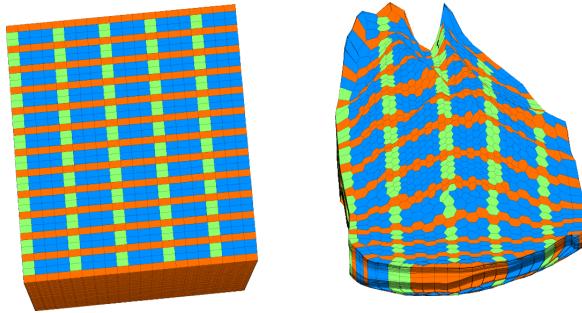


Fig. 3.27. Example of a virtual grid used for fast lookup in a 2.5D PEBI grid. The virtual grid has dimensions $37 \times 20 \times 150$ while the PEBI grid has ik dimensions 1174×150 .

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 $n_f = \text{facePos}(\text{end})-1$.

- `faces`: an $n_f \times 3$ array that gives the global faces connected to a given cell. Specifically, if `faces(i,1)==j`, the face with global number `faces(i,2)` is connected to 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: `West`, `East`, `South`, `North`, `Bottom`, `Top`.

The first column of `faces` 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 `faces` 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:

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

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

$$\text{nodes}(\text{nodePos}(i) : \text{nodePos}(i+1)-1, :)$$
The number of nodes of each face may be computed using the statement `diff(nodePos)`. Likewise, the total number of nodes is given as $n_n = \text{nodePos}(\text{end}) - 1$.
- `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

```
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 3.28 shows⁴ the geometry

⁴ To create the plot in Figure 3.28, 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.

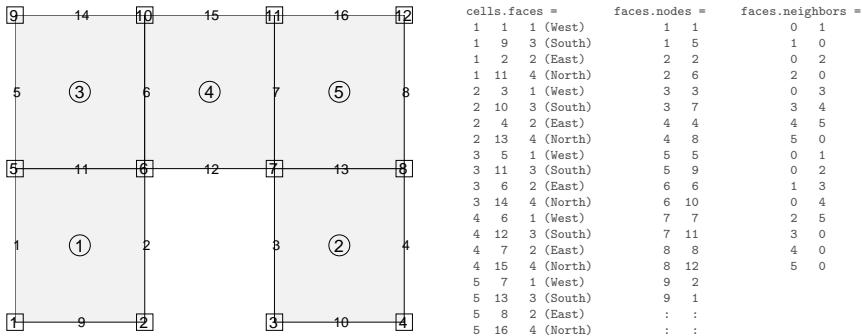


Fig. 3.28. Illustration of the `cell` and `faces` fields of the grid structure: cell numbers are marked by circles, node numbers by squares, and face numbers have no marker.

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:

```
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 3.29 along

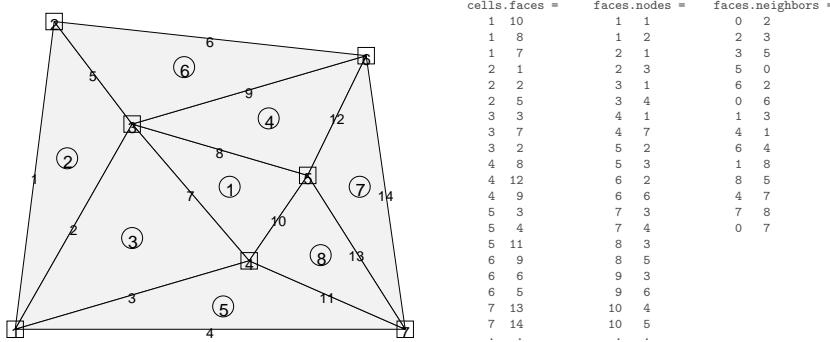


Fig. 3.29. Illustration of the `cell` and `faces` fields of the grid structure: cell numbers are marked by circles, node numbers by squares, and face numbers have no marker. squares.

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 necessarily 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 CO₂ storage in deep saline aquifers using vertically-integrated models that describe the thickness of a supercritical CO₂ 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.

```
[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 3.30. Using `computeGeometry`, we can also compute cell and face centroids, cell areas, and face lengths. Face normals, however, are generally not uniquely defined and are therefore only computed as projections onto the horizontal plane. Likewise, it does not make sense to use this grid together with any of the standard flow and transport solvers in MRST, since neither of these contain discretizations that properly account for processes taking place on non-planar surfaces.

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

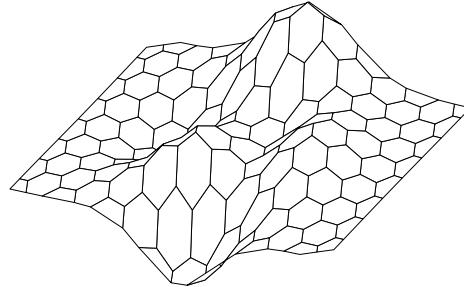


Fig. 3.30. Example of a surface grid: 2D PEBI grid draped over the `peaks` surface.

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 3.31 and 3.32. 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 $\vec{p}(i_1), \dots, \vec{p}(i_m)$ and let $\alpha = (\alpha_1, \dots, \alpha_m)$ denote a multi-index that describes how these points are 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 3.31 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 3.32, which is curved and consists of points $\vec{p}(1), \dots, \vec{p}(5)$ 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 center 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 right in Figures 3.31 and 3.32. The triangles are defined by the points $\vec{p}(\alpha_k)$, $\vec{p}(\alpha_{\text{mod}(k,m)+1})$, and \vec{p}_h

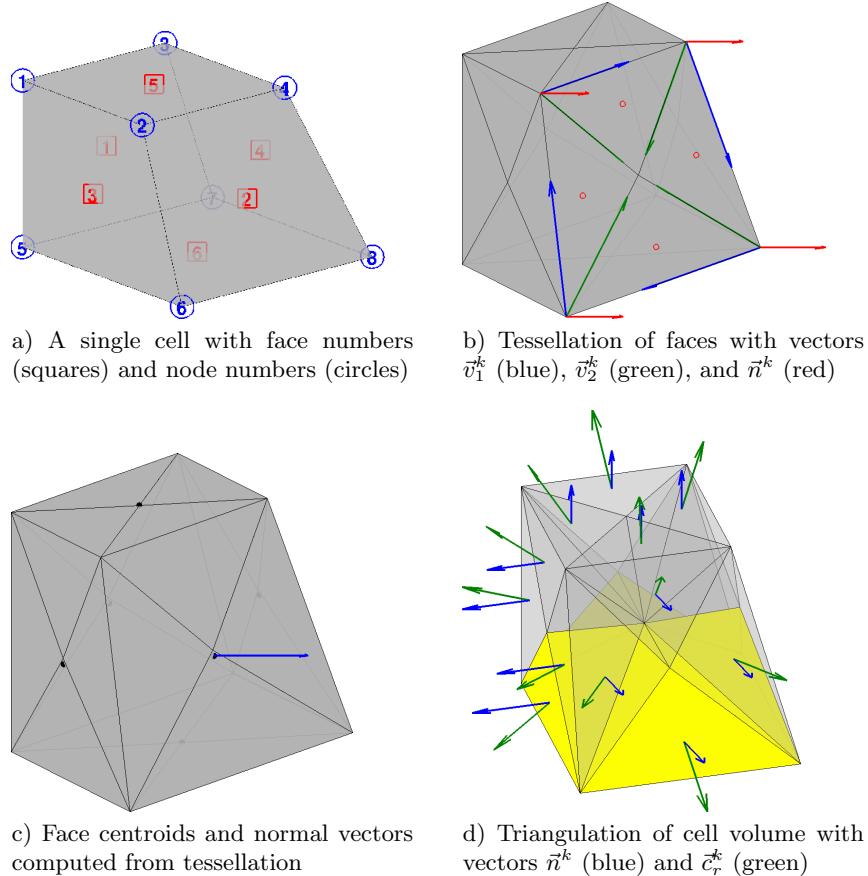


Fig. 3.31. Steps in the computation of geometry information for a single corner-point cell using `computeGeometry`.

for $k = 1, \dots, m$. 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

$$\vec{n}^k = (\vec{p}(\alpha_{\text{mod}(k,m)+1}) - \vec{p}(\alpha_k)) \times (\vec{p}_h - \vec{p}(\alpha_k)) = \vec{v}_1^k \times \vec{v}_2^k$$

$$A^k = \sqrt{\vec{n}^k \cdot \vec{n}^k}.$$

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

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

The result is shown to the lower left in Figures 3.31, 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 3.32.

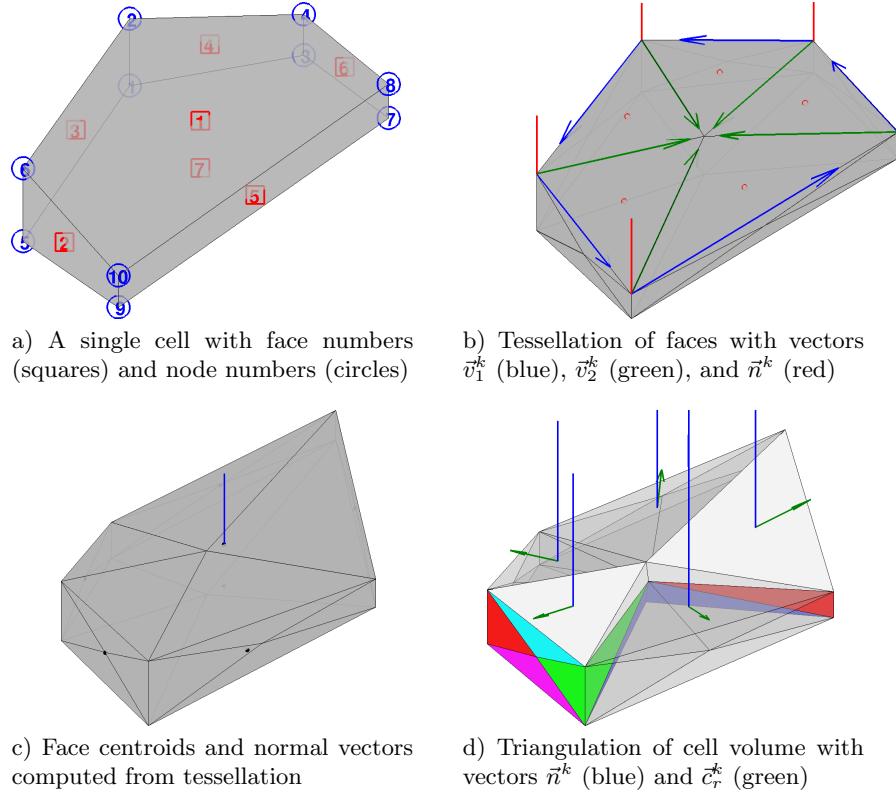


Fig. 3.32. Steps in the computation of geometry information for a single PEBI cell using `computeGeometry`.

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 3.31 and 3.32. For each tetrahedron, we define the vector $\vec{c}_r^k = \vec{p}_c^k - \vec{c}_c$ and compute the volume (which may be negative if the centre point \vec{c}_c lies outside the cell)

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

The triangle normals \vec{n}^k will point outward or inward depending upon the orientation of the points used to calculate them, and to get a correct computation we therefore must modify the triangle normals so that they point outward. Finally, we can define the volume and the centroid of the cell as follows

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

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

COMPUTER EXERCISES:

15. Go back to Exercise 6 in Section 3.1. What would you do to randomly perturb all nodes in the grid except for those that lie on an outer face whose normal vector has no component in the y -direction?
16. Exercise 10 on page 79 extended the function `triangleGrid` from planar triangulations to triangulated surfaces in 3D. Verify that the function `computeGeometry` computes cell areas, cell centroids, face centroids, and face lengths correctly for general 3D triangulated surfaces.
17. How would you write a function that purges all cells that have an invalid vertex (with value NaN) from a grid?

3.5 Examples of more complex grids

To help the user generate test cases, MRST supplies a routines for generating example grids. We have previously encountered `twister`, which perturbs the x and y coordinates in a grid. Likewise, in Chapter 2.5 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 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 3.33.

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 3.34. 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 3.34.

SAIGUP: shallow-marine reservoirs

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 rep-

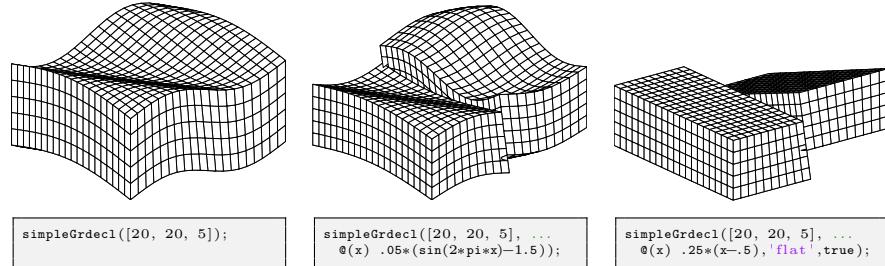


Fig. 3.33. The `simpleGrdecl` routine can be used to produce faulted, two-block grids of different shapes.

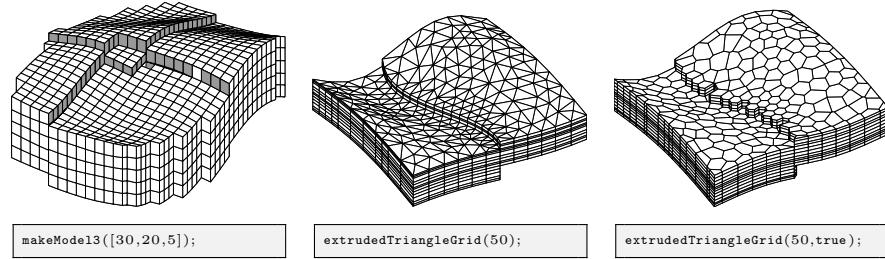


Fig. 3.34. Three different example grids created by the grid example functions `makeModel3` and `extrudedTriangleGrid`.

resentation in more detail and show some examples of how to interact and visualize different features of the grid (see also the last video of the second MRST Jolt on grids and petrophysical data [155]). In Chapter 2.5, 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 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 $22\,500 \text{ m}^3$ and $24\,915 \text{ m}^3$. 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 3.35 shows where these faces appear in the model and a histogram of their areas: the smallest face has an area of $5.77 \cdot 10^{-4} \text{ m}^2$ 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 the model. To this end, we use a simple trick: create a matrix with ones in all

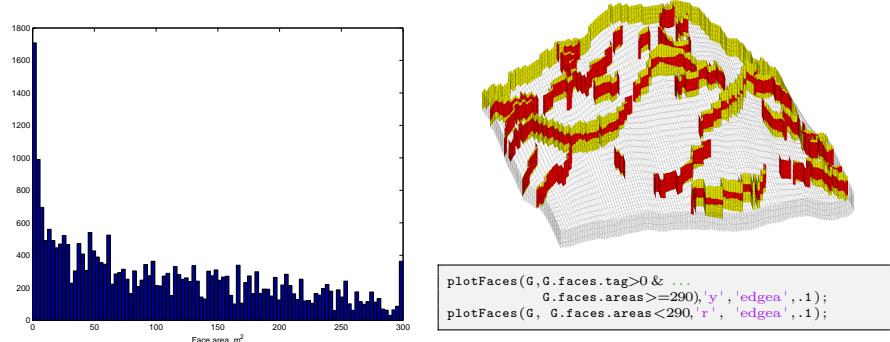


Fig. 3.35. Faces that have been subdivided for the SAIGUP mode. The left plot shows a histogram of the faces areas. The right plot shows all fault faces (yellow) and fault faces having area less than 290 m^2 (red).

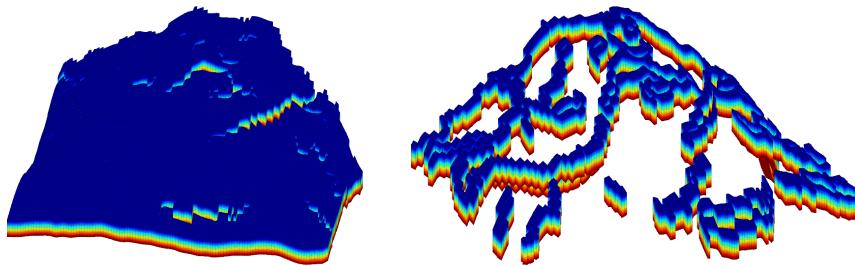


Fig. 3.36. Visualizing the layered structure of the SAIGUP model.

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 3.36. 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 3.36. Let us inspect the fault structure in the lower-right corner

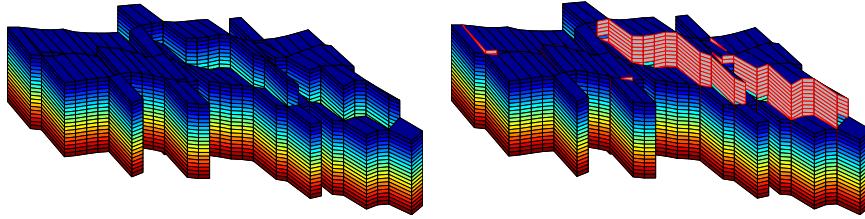


Fig. 3.37. Details from the SAIGUP model showing a zoom of the fault structure in the lower-right corner of the right plot in Figure 3.36. The left plot shows the cells attached to the fault faces, and in the right plot the fault faces have been marked with gray color and red edges.

of the plot. If we disregard using `cutGrdecl` as discussed on page 87, 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 built-in 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));
```

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 boolean numbers that are `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 3.37. 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');
```

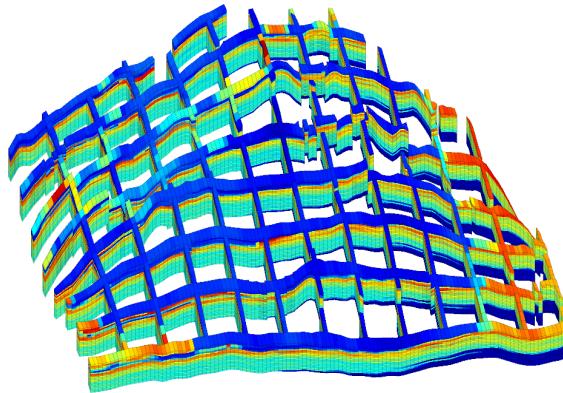


Fig. 3.38. A 'sieve' plot of the porosity in the SAIGUP model. Using this technique, one can more easily see the structure in the interior of the model.

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 throughout the model. That is, different grid types of cells or different grid resolution may be used locally to adapt to well trajectories and flow and

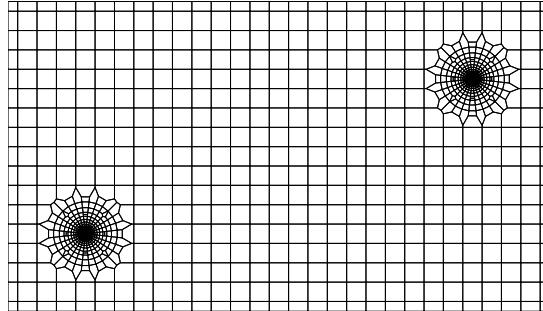


Fig. 3.39. A composite grid consisting of a regular Cartesian mesh with radial refinement around two well positions.

geological constraints, see e.g., [99, 174, 91, 167, 45, 72, 244] and references therein.

You may already have encountered a composite grid if you did Exercise 12 on page 79, where we sought an unstructured grid that adapted to two skew faults and was padded with rectangular cells near the boundary. As another 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 points for a unit refinement, as discussed in Figure 3.25 on page 92

```
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 3.39. 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

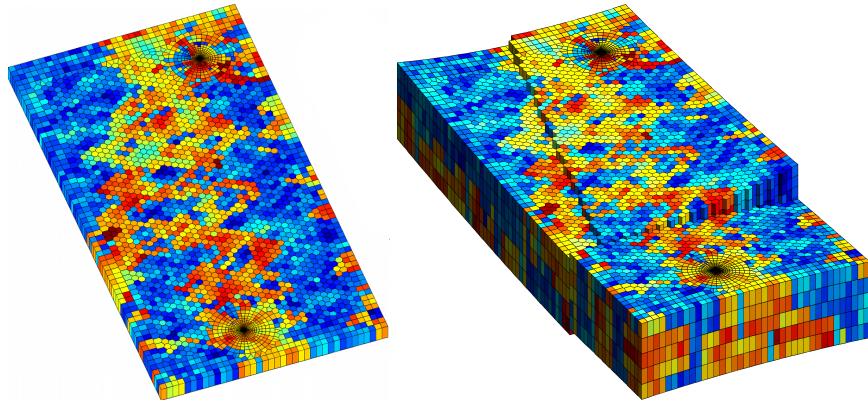


Fig. 3.40. Examples of composite grids. The left plot shows an areal grid consisting of Cartesian, hexagonal, and radial parts. The right plot shows the same grid extruded to 3D with two stair-stepped faults added.

plot in Figure 3.10 on page 74) and can, if necessary be avoided by a more meticulous choice of points.

In the left plot of Figure 3.40, we have combined these two approaches to generate an areal grid consisting of three characteristic components: Cartesian grid cells at the outer boundary, hexagonal cells in the interior, and a radial grid with exponential radial refinement around two wells. The right plot shows a 2.5D in which the areal Voronoi grid has been extruded to 3D along vertical pillars. In addition, structural displacement has been modelled along two areally stair-stepped faults that intersect near the west boundary. Petrophysical parameters have been sampled from layers 40–44 of the SPE 10 data set [62].

3.5.1 Control-point and boundary-aligned grids

To correctly split the reservoir volume into sub-volumes, build flow units with similar or correlated petrophysical properties, and resolve flow pattern, it is important that a volumetric simulation grid conforms as closely as possible fault surfaces and horizons representing the structural architecture of the reservoir. This can be achieved by introducing unstructured grids whose cell interfaces adapt to curved lines in 2D and triangulated surfaces in 3D. Likewise, it is of interest to have grids that can adapt the centroids of certain cells to 2D lines or 3D lines or planes. This is particularly relevant to provide accurate description of well paths and the near-well region. Modern wells are typically long and horizontal and may have complex geometry that consists of multiple branches. Such wells are increasingly becoming the main determining factor for reservoir flow patterns because of their long reach, but also as a result of various techniques for modifying the near-well region to increase injectivity or use of (intelligent) inflow devices to control fluid production.

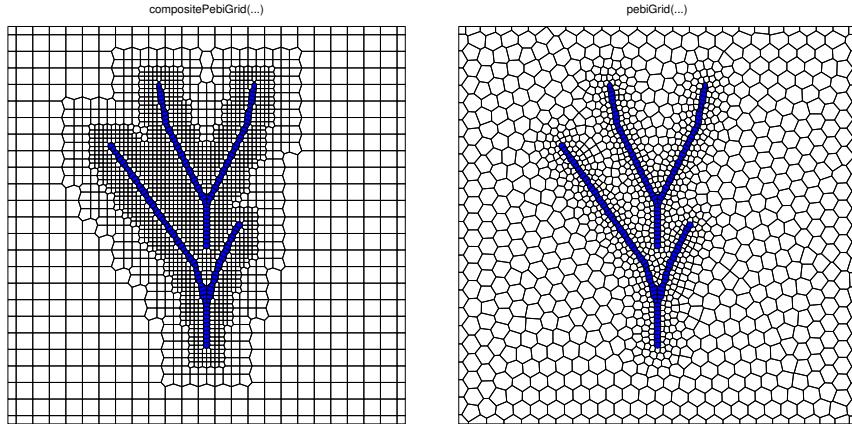


Fig. 3.41. Two different 2D grids adapting to curved well paths. The hybrid grid shown to the left has Cartesian cells away from the well path, with two levels of refinement in the near-well zone. Polygonal cells are used on the transition between refinement levels and to track the well path. The Voronoi grid shown to the right has gradual refinement towards the well and is computed as the dual to a triangulation generated by `DistMesh`.

MRST has one add-on third-party module called `upr`, which offers such adaptivity in the form of 2D Voronoi grids and/or composite grids and fully 3D Voronoi grid. The module was initially developed by one of my previous students as part of his master thesis [39]. I will not go into more details about the underlying methods; the interested reader can find more details, as well as references to other related work, in [39, 206]. Instead, I have included Figures 3.41 to 3.44, which illustrate the flexibility in geometric modeling inherent in such grids.

Multiblock grids

A somewhat different approach to get grids whose geometry and topology vary throughout the physical domain is to use multiblock grids in which different types of structured or unstructured subgrids are glued together. The resulting grid can be non-matching across block interfaces (see e.g., [258, 21, 257]) or have grid lines that are continuous (see e.g., [117, 147]). MRST does not have any grid-factory routine for generating advanced multiblock grids, but offers the function `glue2DGrid` for gluing together rectangular blocks in 2D. These grids can, if necessary, be extruded to 3D by the use of `makeLayeredGrid`. In the following, we will show a few examples of such grids.

As our first example, let us generate a curvilinear grid that has a local refinement at its center as shown in Figure 3.45 (see also Exercise 11 on page 79). To this end, we start by generating three different block types shown in red, green, and blue colors to the left in the figure:

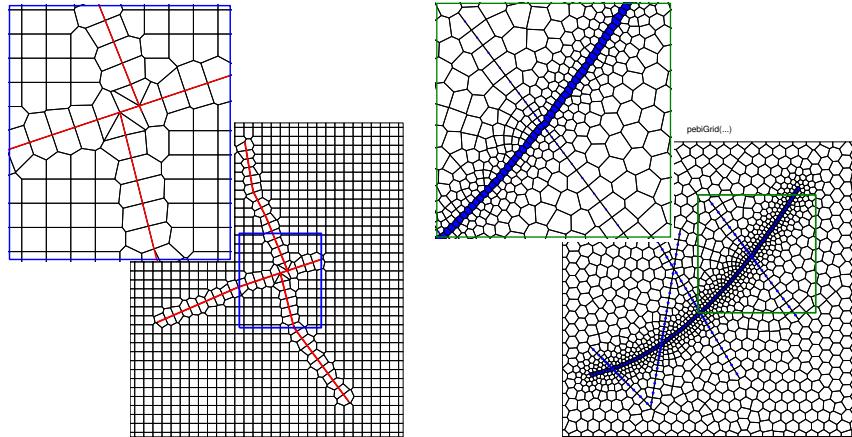


Fig. 3.42. The left grid adapts to three intersecting faults, whereas the grid to the right adapts to both cell and face constraints.

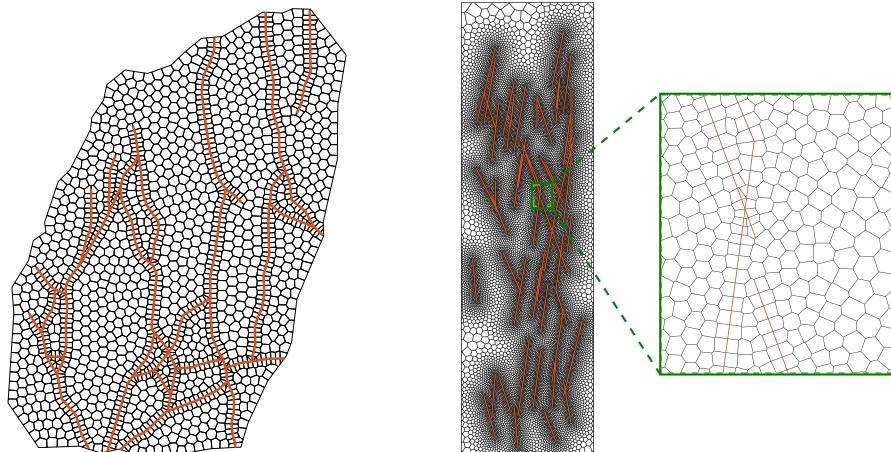


Fig. 3.43. The left grid is an areal description of a complex fault network taken from Branets et al. [42] and is generated as a 2D centroidal Voronoi Diagram (CVD) found by minimizing an energy functional. In the right grid, there are 51 randomly distributed fractures and the resolution is graded towards these.

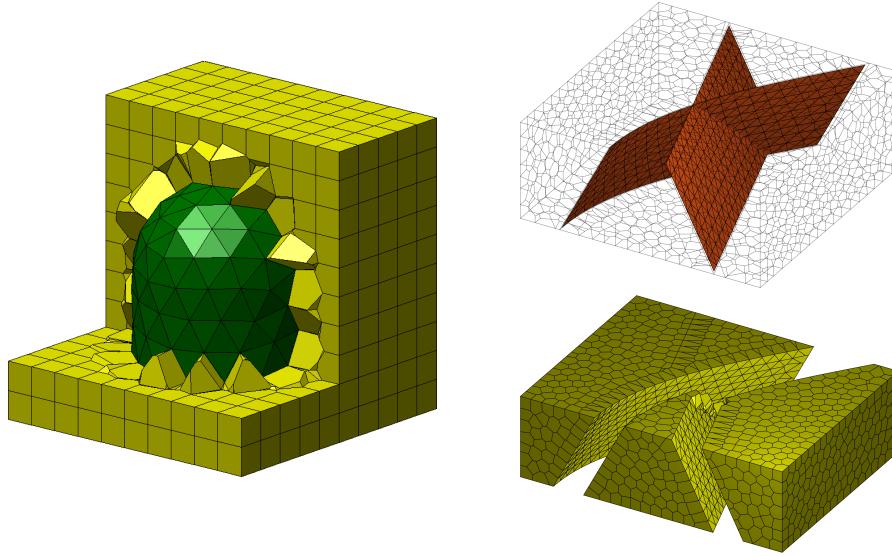


Fig. 3.44. The left grid shows a conceptual model of a salt dome, with triangulated surface extracted from a 3D tetrahedrization generated by `DistMesh`. The plots to the right shows a 3D Voronoi grid adapted to a curved and a planar fault. At the intersection of these, only the curved fault is represented exactly.

```
G1 = cartGrid([ 5  5],[1 1]);
G2 = cartGrid([20 20],[1 1]);
G3 = cartGrid([15  5],[3 1]);
```

Once these are in place, we can simply translate the blocks and glue them together and then apply the `twister` function to make a curvilinear transformation of each grid line:

```
G = glue2DGrid(G1, translateGrid(G2,[1 0]));
G = glue2DGrid(G,  translateGrid(G1,[2 0]));
G = glue2DGrid(G3, translateGrid(G, [0 1]));
G = glue2DGrid(G,  translateGrid(G3,[0 2]));
G = twister(G);
```

Let us now replace the central block by a patch consisting of triangular cells. To this end, we start by regenerating G2

```
[N,M]=deal(10,15);
[x,y] = ndgrid( linspace(0,1,N+1), linspace(0,1,M+1));
x(2:N,2:M) = x(2:N,2:M) + 0.3*randn(N-1,M-1)*max(diff(xv));
y(2:N,2:M) = y(2:N,2:M) + 0.3*randn(N-1,M-1)*max(diff(yv));
G2 = computeGeometry(triangleGrid([x(:) y(:)]));
```

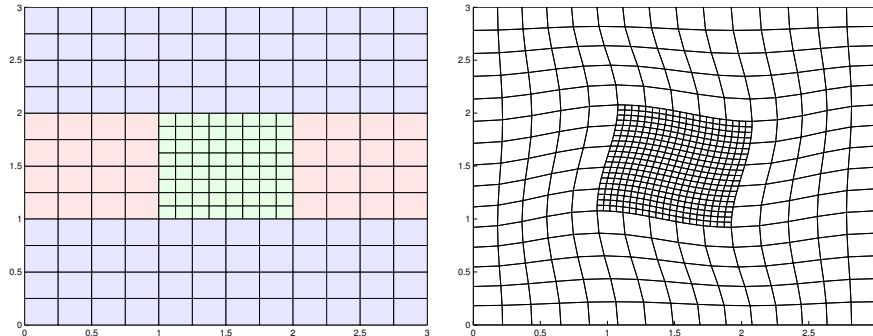


Fig. 3.45. Using a multiblock approach to construct a rectilinear grid with refinement.

The `glue2DGrid` routine relies on face tags as explained in [3.4 on page 95](#) that can be used to identify the external faces that are facing East, West, North, and South. Generally, such tags does not make much sense for triangular grids and are therefore not supplied. However, to be able to find the correct interface to glue together, we need to supply tags on the perimeter of the triangular patch, where the normal vectors follow the axial directions and tags therefore make sense. To this end, we start by computing the true normal vectors:

```
hf = G2.cells.faces(:,1);
hf2cn = gridCell1No(G2);
sgn = 2*(hf2cn == G2.faces.neighbors(hf, 1)) - 1;
N = bsxfun(@times, sgn, G2.faces.normals(hf,:));
N = bsxfun(@rdivide, N, G2.faces.areas(hf,:));
n = zeros(numel(hf),2); n(:,1)=1;
```

Then, all interfaces that face the East are those whose dot-product with the vector $(1, 0)$ is identical to -1 :

```
G2.cells.faces(:,2) = zeros(size(hf));
i = sum(N.*n,2)==-1; G2.cells.faces(i,2) = 1;
```

Similarly, we can identify all the interfaces facing the West, North, and South. The left plot in Figure 3.46 shows the resulting multiblock grid. Likewise, the right plot shows another multiblock grid where the central refinement is the dual to the triangular patch and $G3$ has been scaled in the y -direction and refined in the x -direction so that the grid lines are no longer matching with the grid lines of $G1$.

As a last example, let us use this technique to generate a 3D multiblock grid that consists of three blocks in the vertical direction

```
G = glue2DGrid(G1, translateGrid(G2, [0 1]));
G = glue2DGrid(G, translateGrid(G1, [0 2]));
G = makeLayeredGrid(G, 5);
G.nodes.coords = G.nodes.coords(:,[3 1 2]);
```

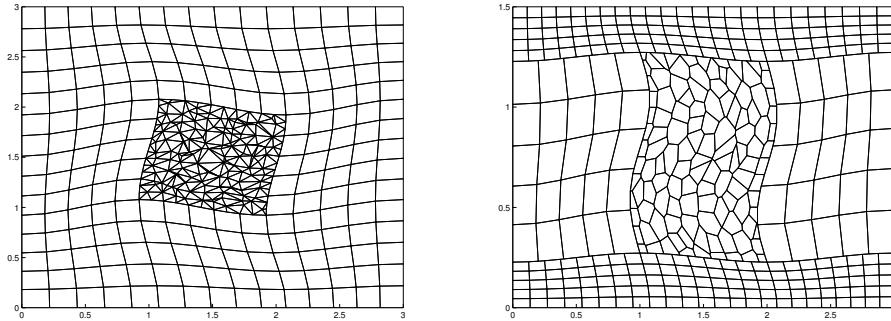


Fig. 3.46. Using a multiblock approach to construct rectilinear grid with triangular and polygonal refinements.

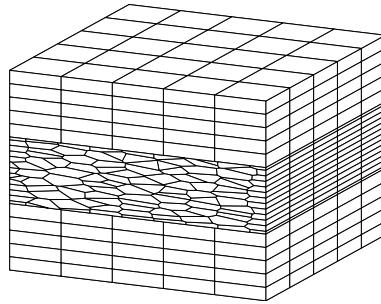


Fig. 3.47. An example of a 3D multiblock grid.

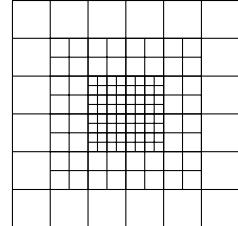
That is, we first generate an areal grid in the xy -plane, extrude it to 3D along the z direction, and then permute the axis so that their relative orientation is correctly preserved (notice that simply flipping $[1\ 2\ 3] \rightarrow [1\ 3\ 2]$, for instance, will *not* create a functional grid).

COMPUTER EXERCISES:

17. How would you populate the grids shown in Figures 3.33 and 3.34 with petrophysical properties so that spatial correlation and displacement across the fault(s) is correctly accounted for? As an illustrative example, you can try to sample petrophysical properties from the SPE 10 data set.
18. Select at least one of the models in the data sets `BedModels1` or `BedModel2` and try to find all inactive cells and then all cells that do not have six faces. Hint: it may be instructive to visualize these models both in physical space and in index space.
19. Extend your function from Exercise 12 on page 79 to also include radial refinement in near-well regions as shown in Figure 3.39.
20. Make a grid similar to the one shown to the right in Figure 3.40. Hint: although it is not easy to see, the grid is matching across the fault, which

means that you can use the method of fictitious domain to make the fault structure.

21. As pointed out in Exercise 11 on page 79, MRST does not yet have a grid factory routine to generate structured grids with local nested refinement as shown in the figure to the right. While it is not very difficult to generate the necessary vertices if each refinement patch is rectangular and matches the grid cells on the coarser level, building up the grid structure may prove to be a challenge. Try to develop an efficient algorithm and implement in MRST.



Part II

Single-Phase Flow

4

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 1.4 where we showed how to use MRST to compute vertical equilibrium inside a gravity column, and then in Section 2.4.2, 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.

4.1 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 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 4.1 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] and the

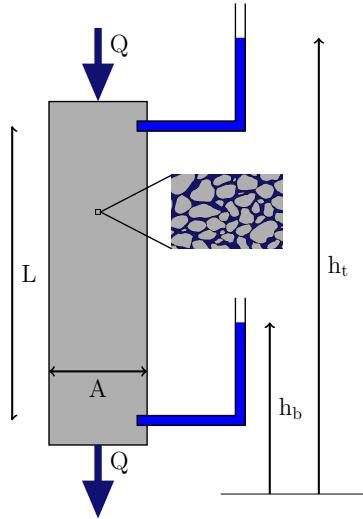


Fig. 4.1. Conceptual illustration of Darcy's experiment.

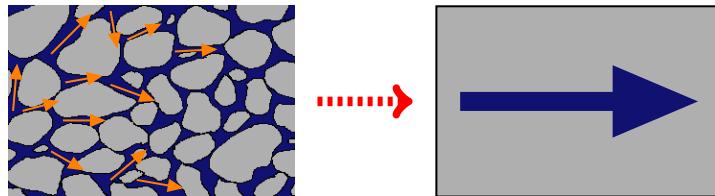


Fig. 4.2. The macroscopic Darcy velocity represents an average of microscopic fluid fluxes.

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

$$\frac{Q}{A} = \kappa \frac{h_t - h_b}{L} \quad (4.1)$$

which was presented in 1856 as an appendix to [67] entitled “Determination of the laws of flow of water through sand” and is what we today call Darcy’s law. In (4.1), 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/s²] is the gravitational acceleration, μ [kg/ms] is the dynamic viscosity, and K [m²] 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-

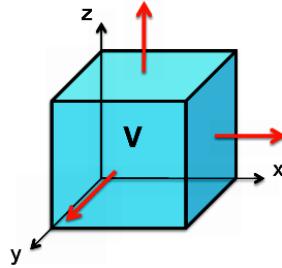


Fig. 4.3. Illustration of a control volume Ω on which one can apply the principle of conservation to derive macroscopic continuity equations.

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 2.3.2. 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,

$$\vec{v} = -\frac{\kappa}{\mu}(\nabla p - g\rho\nabla z), \quad (4.2)$$

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 [112] and later from Stokes flow by Whitaker [259]. The observant reader will notice that Darcy's law (4.2) 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.

4.2 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 REVs as discussed in the previous section and then look at a control volume as shown in Figure 4.3. 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,

$$\frac{\partial}{\partial t} \int_{\Omega} \phi \rho d\vec{x} + \int_{\partial\Omega} \rho \vec{v} \cdot \vec{n} ds = \int_{\Omega} \rho q d\vec{x}, \quad (4.3)$$

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

$$\int_{\Omega} \left[\frac{\partial}{\partial t} \phi \rho + \nabla \cdot (\rho \vec{v}) \right] d\vec{x} = \int_{\Omega} \rho q d\vec{x}. \quad (4.4)$$

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

$$\frac{\partial(\phi\rho)}{\partial t} + \nabla \cdot (\rho \vec{v}) = \rho q. \quad (4.5)$$

Equation (4.5) 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 2.4.1 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,

$$\frac{dV}{V} = \frac{1}{V} \left(\frac{\partial V}{\partial p} \right)_T dp + \frac{1}{V} \left(\frac{\partial V}{\partial T} \right)_p dT, \quad (4.6)$$

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, $V dp = \rho dV$, and (4.6) can be written in the equivalent form

$$\frac{d\rho}{\rho} = \frac{1}{\rho} \left(\frac{\partial \rho}{\partial p} \right)_T dp + \frac{1}{\rho} \left(\frac{\partial \rho}{\partial T} \right)_p dT = c_f dp + \alpha_f dT, \quad (4.7)$$

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 (4.7) simplifies to

$$c_f = \frac{1}{\rho} \frac{d\rho}{dp} = \frac{d \ln(\rho)}{dp}. \quad (4.8)$$

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 (4.5), we obtain the following parabolic equation for the fluid pressure

$$c_t \phi \rho \frac{\partial p}{\partial t} - \nabla \cdot \left[\frac{\rho K}{\mu} (\nabla p - g \rho \nabla z) \right] = \rho q, \quad (4.9)$$

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 [210, Chap. 1], [60, 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 $c_t = 0$), (4.9) simplifies to an elliptic equation with variable coefficients,

$$-\nabla \cdot \left[\frac{K}{\mu} \nabla (p - g \rho z) \right] = q. \quad (4.10)$$

If we introduce the fluid potential, $\Phi = p - g \rho z$, (4.10) can be recognized as the (generalized) Poisson's equation $-\nabla \cdot K \nabla \Phi = q$ or as the Laplace equation $\nabla \cdot K \nabla \Phi = 0$ 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 $\mathcal{L} = \nabla \cdot K \nabla$, 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, (4.8) can be integrated from a known density ρ_0 at a pressure datum p_0 to give the following equation,

$$\rho(p) = \rho_0 e^{c_f(p-p_0)} \quad (4.11)$$

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 (4.8), 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 (4.2). Inserting the result into (4.5) gives us the following continuity equation

$$\frac{\partial \rho}{\partial t} - \frac{1}{\mu \phi c_f} \nabla \cdot (K \nabla \rho - c_f g \rho^2 K \nabla z) = \rho q, \quad (4.12)$$

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,

$$\frac{\partial \rho}{\partial t} = \frac{1}{\mu \phi c_f} \nabla \cdot (\mathbf{K} \nabla \rho). \quad (4.13)$$

Slightly compressible flow

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

$$\rho = \rho_0 [1 + c_f(p - p_0)]. \quad (4.14)$$

We further assume that ϕ has a similar functional dependence and that μ is constant. For simplicity, we also assume that g and q are both zero. Then, we can simplify (4.9) as follows:

$$[(c_r + c_f)\phi\rho] \frac{\partial p}{\partial t} = \frac{c_f \rho}{\mu} \nabla p \cdot \mathbf{K} \nabla p + \frac{\rho}{\mu} \nabla \cdot (\mathbf{K} \nabla p)$$

If c_f is sufficiently small, in the sense that $c_f \nabla p \cdot \mathbf{K} \nabla p \ll \nabla \cdot (\mathbf{K} \nabla p)$, we can neglect the first term on the right-hand side to derive a linear equation similar to (4.13) for the fluid pressure (introducing total compressibility $c = c_r + c_f$)

$$\frac{\partial p}{\partial t} = \frac{1}{\mu \phi c} \nabla \cdot (\mathbf{K} \nabla p). \quad (4.15)$$

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,

$$pV = nRT, \quad \rho = p(\gamma - 1)e. \quad (4.16)$$

In the first form, T is temperature, V is volume, R is the gas constant ($8.314 \text{ J K}^{-1}\text{mol}^{-1}$), 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 heat at constant pressure and constant volume, and e is the specific internal energy (internal energy per unit mass). In either case, it follows from (4.8) 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 (4.16) into (4.9) gives

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

from which it follows that

$$\phi \mu \frac{\partial p}{\partial t} = \nabla \cdot (p \mathbf{K} \nabla p) \quad \Leftrightarrow \quad \frac{\phi \mu}{p} \frac{\partial p^2}{\partial t} = \nabla \cdot (\mathbf{K} \nabla p^2). \quad (4.17)$$

Equation of state

Equations (4.11), (4.14), and (4.16) 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 equation of state

$$\begin{aligned} p &= \frac{RT}{V_m - b} - \frac{a}{\sqrt{T} V_m (V_m + b)}, \\ a &= \frac{0.42748 R^2 T_c^{5/2}}{p_c}, \quad b = \frac{0.08662 R T_c}{p_c}, \end{aligned} \quad (4.18)$$

the modified version called Redlich–Kwong–Soave

$$\begin{aligned} p &= \frac{RT}{V_m - b} - \frac{a\alpha}{\sqrt{T} V_m (V_m + b)}, \\ a &= \frac{0.427 R^2 T_c^2}{p_c}, \quad b = \frac{0.08664 R T_c}{p_c}, \\ \alpha &= [1 + (0.48508 + 1.55171\omega - 0.15613\omega^2)(1 - \sqrt{T/T_c})]^2, \end{aligned} \quad (4.19)$$

as well as the Peng–Robinson equation of state,

$$\begin{aligned} p &= \frac{RT}{V_m - b} - \frac{a\alpha}{V_m^2 + 2bV_m - b^2}, \\ a &= \frac{0.4527235 R^2 T_c^2}{p_c}, \quad b = \frac{0.077796 R T_c}{p_c}, \\ \alpha &= [1 + (0.37464 + 1.54226\omega - 0.26992\omega^2)(1 - \sqrt{T/T_c})]^2. \end{aligned} \quad (4.20)$$

Here, 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

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

the Redlich–Kwong–Soave equation (4.19) and the Peng–Robinson equation (4.20) can be written in alternative polynomial forms,

$$0 = Z^3 - Z^2 + Z(A - B - B^2) - AB, \quad (4.21)$$

$$0 = Z^3 - (1 - B)Z^2 + (A - 2B - 3B^2)Z - (AB - B^2 - B^3), \quad (4.22)$$

which illustrates why they are called cubic equations of state.

4.3 Auxiliary conditions and equations

The governing equations for single-phase flow discussed above are all parabolic, except for the incompressible case in which the governing equation is elliptic. For the solution to be well-posed¹ 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.

4.3.1 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,

$$\vec{v} \cdot \vec{n} = 0 \quad \text{for } \vec{x} \in \partial\Omega. \quad (4.23)$$

With no-flow boundary conditions, any pressure solution of (4.10) 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 provides external pressure support, which can be modeled in terms of a Dirichlet condition of the form

$$p(\vec{x}) = p_a(\vec{x}, t) \quad \text{for } \vec{x} \in \Gamma_a \subset \partial\Omega. \quad (4.24)$$

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,

$$\vec{v} \cdot \vec{n} = u_a(\vec{x}, t) \quad \text{for } \vec{x} \in \Gamma_a \subset \partial\Omega. \quad (4.25)$$

Combinations of these conditions are used when studying parts of a reservoir (e.g., sector models). There are also cases, e.g., when describing groundwater

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

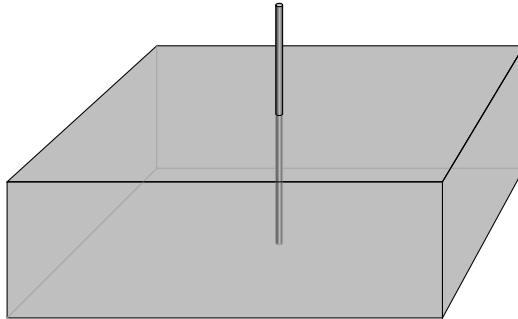


Fig. 4.4. Illustration of a well inside a grid cell. The proportions are not fully to scale: whereas the diameter of a well varies from 5 to 40 inches, a grid block may extend from tens to hundreds of meters in the lateral direction and from a few decimeters to ten meters in the vertical direction.

systems or CO₂ 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 5.1.4. In the compressible case in (4.9), 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 1.4, and hence be given by the ordinary differential equation,

$$\frac{dp}{dz} = \rho g, \quad p(z_0) = p_0. \quad (4.26)$$

4.3.2 Injection and production wells

In a typical reservoir simulation, the inflow and outflow in wells occur on a subgrid scale. In most discretized flow models, the pressure is modelled using a single pressure value inside each grid cell. The size of each grid cell must therefore be chosen so small that the pressure variation inside the cell can be approximated accurately in terms of its volumetric average. Far away from wells, the spatial variations in pressure tend to be relatively slow, at least in certain directions, and one can therefore choose cell sizes in the order of tens or hundreds of meters, which is a reasonable size compared with the extent of the reservoir. Near the well, however, the pressure will have large variations over short distances, and to compute a good approximation of these pressure variations, one would need grid cells that are generally smaller than what is computationally tractable. As a result, one with a setup similar to what is illustrated in Figure 4.4, where the radius of the well is typically between 1/100 and 1/1000 of the horizontal dimensions of the grid cell. The largest percentage of the pressure drop associated with a well occurs near the well and the pressure at the well radius will thus deviate significantly from the volumetric pressure average inside the cell. Special analytical models are

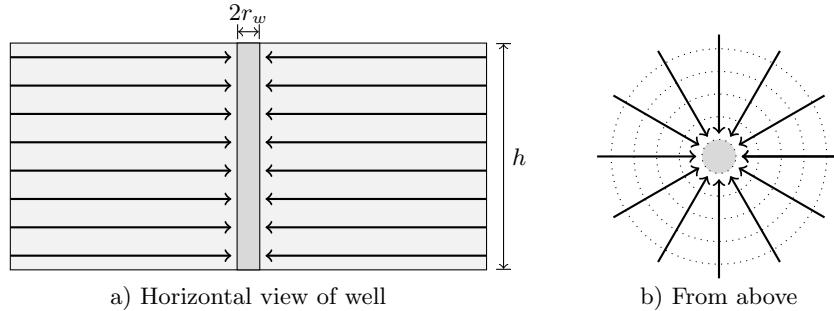


Fig. 4.5. Radial flow used to develop well model

therefore developed to represent the subgrid variations in the particular flow patterns near wells.

Normally, fluids are injected from a well at either constant *surface rate* or at constant *bottom-hole pressure*, which is also called wellbore flowing pressure and refers to the pressure at a certain point inside the wellbore. Similarly, fluids are produced at constant bottom-hole pressure or at constant surface liquid rate. The main purpose of a well model is then to accurately compute the pressure at well radius when the injection or production rate is known, or to accurately compute the flow rate in and out of the reservoir when the pressure at well radius is known. The resulting relation between the bottom-hole pressure and surface flow rate is often called the 'inflow-performance relation' or IPR.

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

$$q_o = J(p_R - p_w), \quad (4.27)$$

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 pressure p_w in the wellbore. 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 (4.27) 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 over a height h and is open in the radial direction. Fluids are assumed to only flow in the radial direction and the outer boundary is circular, see Figure 4.5. In other words, we assume a pseudo-steady, radial

flow. From mass conservation written in cylinder coordinates, we have

$$\frac{1}{r} \frac{\partial(rvB^{-1})}{\partial r} = 0 \quad \rightarrow \quad v = BC/r.$$

The constant C is determined by integrating around a small cylinder surrounding the sink term,

$$q = \oint B^{-1} \vec{v} \cdot \vec{n} ds = -2\pi h C.$$

Inserting this into Darcy's law in cylinder coordinates, we have

$$v = -\frac{qB}{2\pi rh} = -\frac{K}{\mu} \frac{dp}{dr}.$$

Even if several different flow patterns can be expected when fluids flow toward a vertical wellbore, two-dimensional radial flow is considered to be the most representative for vertical oil and gas wells.

We now integrate this equation from the wellbore r_w and to the drainage boundary r_d where the pressure is constant

$$2\pi K h \int_{p_w}^{p_d} \frac{1}{q\mu B} dp = \int_{r_w}^{r_d} \frac{1}{r} dr.$$

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 $p_{avg} = (p_w + p_d)/2$. For convenience, we drop the subscript in the following. This gives us the pressure as a function of radial distance

$$p_d = p_w + \frac{q\mu B}{2\pi K h} \ln(r_d/r_w). \quad (4.28)$$

To close the system, we need to know the location of the drainage boundary $r = r_d$ 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 p_a is found at $r = 0.472r_d$. Hence,

$$q = \frac{2\pi K h}{\mu B (\ln(r_d/r_w) - 0.75)} (p_a - p_w). \quad (4.29)$$

For an injector, we likewise obtain

$$q = \frac{2\pi K h}{\mu B (\ln(r_d/r_w) - 0.75)} (p_w - p_a).$$

The above relation (4.29) was developed for an ideal well under several simplifying assumptions: homogeneous and isotropic formation of constant thickness, clean wellbore, etc. In practice, a well will rarely experience these ideal

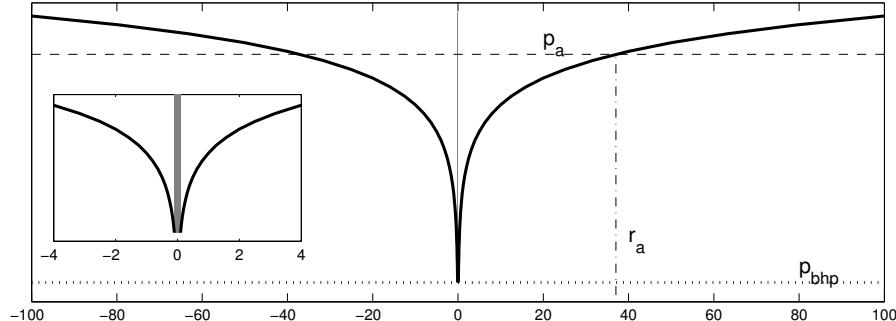


Fig. 4.6. Illustration of the pressure distribution inside a cell computed from (4.28) assuming the well is producing fluids from an infinite domain. Here, p_a is the volumetric pressure average and r_a is the radius at which this value is found. The inset shows a zoom of the near-well zone.

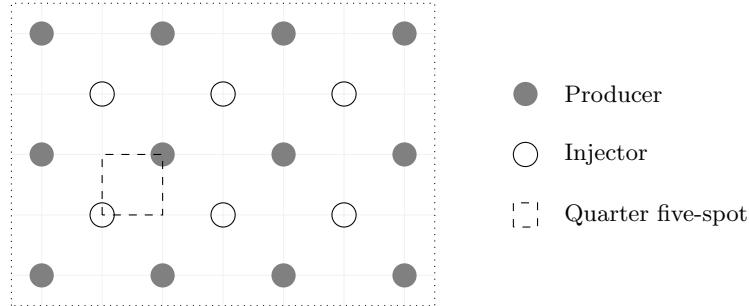
conditions. Typically the permeability is altered close to the wellbore under drilling and completion, the well will only be partially completed, and so on. The actual pressure performance will therefore deviate from (4.29). To model this, it is customary to include a *skin factor* S to account for extra pressure loss due to alterations in the inflow zone. The resulting equation is

$$q = \frac{2\pi K h}{\mu B (\ln(r_d/r_w) - 0.75 + S)} (p_a - p_w). \quad (4.30)$$

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 pressure p_R . Analytical solutions are generally not known since real reservoirs have complicated geometries and irregular boundaries. Well models are therefore developed using highly idealized reservoir geometries. One such example is the so-called *repeated five-spot pattern*, which consists of a thin, infinitely large, horizontal reservoir with a staggered pattern of injection and production wells as shown in Figure 4.7 that repeats itself to infinity in all directions. The name comes from the fact that each injector is surrounded by four producers, and vice versa, hence creating tiles of five-spot patterns. If all wells operate at equal rates, the flow pattern has certain symmetries and it is common to only consider a quarter of the five spot, as shown in Figure 4.7, subject to no-flow boundary conditions. An analytical solution for the pressure drop between the injection and production wells was developed by Muskat [185],

$$\Delta p = \frac{q \mu B}{\pi K h} \left(\ln(d/r_w) - C \right), \quad (4.31)$$

**Fig. 4.7.** Excerpts of a repeated five-spot pattern.

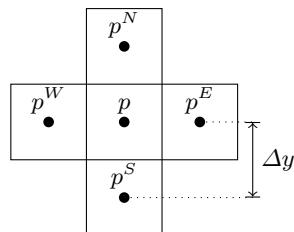
where d is the distance between the wells, and C is given by an infinite series. Muskat [185] originally used $C = 0.6190$, but a more accurate value, $C = 0.61738575$, was later derived by Peaceman [209], who used (4.31) to determine an equivalent radius r_e at which the cell pressure is equal to the analytical pressure. Assuming isotropic permeabilities, square grid blocks, single-phase flow, and a well at a center of an interior block, Peaceman [211] showed that the equivalent radius is

$$r_e \approx 0.20788\sqrt{\Delta x \Delta y}$$

for the two-point discretization that will be discussed in more detail in Section 4.4.1. To give you an idea, let us present a somewhat simplified derivation in the case without gravity. As above, we approximating the pressure-dependent viscosity and formation-volume factor by a suitable average and drop the subscript. For a single-layered square grid (Figure 4.8), the two-point scheme then simplifies to a standard five-point difference scheme

$$-\frac{Kh}{\mu B} [4p - p^W - p^N - p^W - p^S] = q.$$

The solution is symmetric for the repeated five-spot pattern, so that $p^E = p^N = p^W = p^S$ (which explains why μB could be factored out). Hence, we can eliminate the last three pressure values and write the five-point stencil as

**Fig. 4.8.** Five-point stencil on a grid with square grid blocks ($\Delta x = \Delta y$).

$$p = p^E - \frac{q\mu B}{4Kh}.$$

Next, we assume that the radial well solution is an accurate approximation to the pressure values in the neighboring cells,

$$p^E = p_w + \frac{q\mu B}{2\pi Kh} \ln(\Delta x/r_w)$$

Combining the two equations, we have

$$\begin{aligned} p &= p_w + \frac{q\mu B}{2\pi Kh} \ln(\Delta x/r_w) - \frac{q\mu B}{4Kh} \\ &= p_w + \frac{q\mu B}{2\pi Kh} \left[\ln(\Delta x/r_w) - \frac{\pi}{2} \right] = p_w + \frac{q\mu B}{2\pi Kh} \ln(e^{-\pi/2} \Delta x/r_w). \end{aligned}$$

Now, $\exp(-\pi/2) \approx 0.20788$, and hence we have derived Peaceman's classical result.

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 [14, 85, 7]. For anisotropic permeabilities $K = \text{diag}(K_x, K_y, K_z)$, the permeability is replaced by an effective permeability

$$K_e = \sqrt{K_x K_y}, \quad (4.32)$$

which follows by a coordinate transformation $x \rightarrow \hat{x}\sqrt{K_x}$, etc. For a grid with rectangular grid blocks ($\Delta x \neq \Delta y$), the equivalent radius reads [209]

$$r_e = 0.28 \frac{\left(\sqrt{K_y/K_x} \Delta x^2 + \sqrt{K_x/K_y} \Delta y^2 \right)^{1/2}}{\left(K_y/K_x \right)^{1/4} + \left(K_x/K_y \right)^{1/4}}, \quad (4.33)$$

The same formulas apply to horizontal wells, with the necessary permutations; that is, for a well in the x -direction $\Delta x \rightarrow \Delta z$ and $K_x \rightarrow K_z$.

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

$$q_i = \frac{2\pi h K_e}{\mu_i B_i [\ln(r_e/r_w) + S]} (p_R - p_w - \rho_i(z - z_{bh})g), \quad (4.34)$$

where K_e is given by (4.32) and r_e is given by (4.33). 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.

At this point we should add a word of caution. The equivalent radius of a numerical method generally depends on how the method approximates the pressure inside the grid cell containing the well perforation. The formulas given above are strictly seen only valid if you use the specific two-point discretization they were developed for. When using another discretization method, you may have to compute other values for the equivalent radius, e.g., as discussed in [153, 164].

4.3.3 Field lines and time-of-flight

Equation (4.10) 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

$$\frac{d\vec{x}}{dr} \times \vec{v}(\vec{x}, \hat{t}) = 0, \quad \text{or equivalently,} \quad \frac{d\vec{x}}{dr} = \frac{\vec{v}(\hat{t})}{|\vec{v}(\hat{t})|}. \quad (4.35)$$

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

$$\frac{d\vec{y}}{dt} = \vec{v}(\vec{y}, t), \quad \vec{y}(t_0) = \vec{y}_0. \quad (4.36)$$

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

$$\frac{d\vec{z}}{dt} = \vec{v}(\vec{z}, t), \quad \vec{z}(s) = \vec{z}_0. \quad (4.37)$$

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 than pathlines and streaklines. Moreover, rather than using the arc length r to parametrize streamlines, it is common to introduce an alternative parametrization called

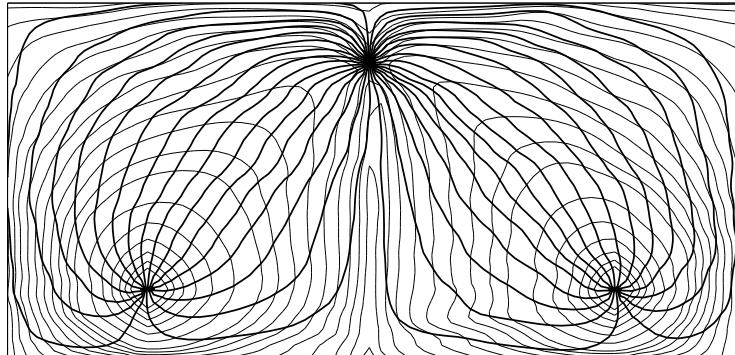


Fig. 4.9. Illustration of time-of-flight, shown as gray isocontour lines, and streamlines shown as thick black lines.

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

$$\tau(r) = \int_0^r \frac{\phi(\vec{x}(s))}{|\vec{v}(\vec{x}(s))|} ds, \quad (4.38)$$

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}/ϕ). 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 [68, 69]

$$\vec{v} \cdot \nabla \tau = \phi. \quad (4.39)$$

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

4.3.4 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 (4.5),

$$\frac{\partial(\phi C)}{\partial t} + \nabla \cdot (\vec{v} C) = q_C. \quad (4.40)$$

Communication patterns within a reservoir can be determined by simulating the evolution of artificial, non-diffusive tracers whose concentration does not change upon fluid compression or expansion. 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 solution approached at steady-state conditions from the non-conservative equation,

$$\vec{v} \cdot \nabla C = q_C, \quad C|_{\text{inflow}} = 1. \quad (4.41)$$

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 [30], for which $\vec{v} = \nabla\psi \times \nabla\chi$. In the streamline coordinates (τ, ψ, χ) , the gradient operator is expressed as

$$\nabla_{(\tau, \psi, \chi)} = (\nabla\tau) \frac{\partial}{\partial\tau} + (\nabla\psi) \frac{\partial}{\partial\psi} + (\nabla\chi) \frac{\partial}{\partial\chi}. \quad (4.42)$$

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 (4.39) that

$$\vec{v} \cdot \nabla_{(\tau, \psi, \chi)} = (\vec{v} \cdot \nabla\tau) \frac{\partial}{\partial\tau} = \phi \frac{\partial}{\partial\tau}. \quad (4.43)$$

Therefore, the coordinate transformation $(x, y, z) \rightarrow (\tau, \psi, \chi)$ will reduce the three-dimensional transport equation (4.40) to a family of one-dimensional transport equations along each streamline [68, 133], which for incompressible flow reads

$$\frac{\partial C}{\partial t} + \frac{\partial C}{\partial\tau} = 0. \quad (4.44)$$

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 (4.44). Here, the response is given as (see [68]),

$$C(t) = C_0(t - \tau), \quad (4.45)$$

which is easily verified by inserting the expression into (4.44) and the fact that the solution is unique [107]. For the special case of continuous and constant injection, the solution is particularly simple

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

4.4 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 6, 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.

4.4.1 Two-point flux-approximation

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

$$\nabla \cdot \vec{v} = q, \quad \vec{v} = -K \nabla p, \quad \text{in } \Omega \subset \mathbb{R}^d. \quad (4.46)$$

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

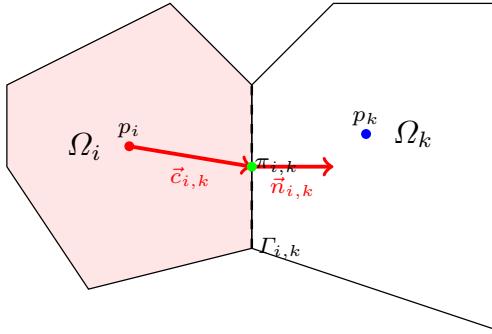


Fig. 4.10. Two cells used to define the two-point finite-volume discretization of the Laplace operator.

To develop a finite-volume discretization for (4.46), we start by rewriting the equation in integral form using a single cell Ω_i in the discrete grid as control volume

$$\int_{\partial\Omega_i} \vec{v} \cdot \vec{n} ds = \int_{\Omega_i} q d\vec{x}. \quad (4.47)$$

This is a simpler form of (4.3), where that the accumulation term has disappeared because ϕ and ρ are independent of time and the constant ρ has been eliminated.

Equation (4.47) ensures 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,

$$v_{i,k} = \int_{\Gamma_{i,k}} \vec{v} \cdot \vec{n} ds, \quad \Gamma_{i,k} = \partial\Omega_i \cap \partial\Omega_k. \quad (4.48)$$

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 (4.48) is approximated by the midpoint rule, we use Darcy's law to write the flux as

$$v_{i,k} \approx A_{i,k} \vec{v}(\bar{x}_{i,k}) \cdot \vec{n}_{i,k} = -A_{i,k} (\nabla p)(\bar{x}_{i,k}) \cdot \vec{n}_{i,k}, \quad (4.49)$$

where $\bar{x}_{i,k}$ 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 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 4.10)

$$v_{i,k} \approx A_{i,k} K_i \frac{(p_i - \pi_{i,k}) \vec{c}_{i,k}}{|\vec{c}_{i,k}|^2} \cdot \vec{n}_{i,k} = T_{i,k}(p_i - \pi_{i,k}). \quad (4.50)$$

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,

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

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

$$v_{ik} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1} (p_i - p_k) = T_{ik}(p_i - p_k). \quad (4.51)$$

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 (4.23) or (4.25), or know the interface pressure for Dirichlet boundary conditions (4.24).

By inserting the expression for v_{ik} into (4.47), we see that the TPFA scheme for (4.46), in compact form, seeks a set of cell averages that satisfy the following system of equations

$$\sum_k T_{ik}(p_i - p_k) = q_i, \quad \forall \Omega_i \subset \Omega \quad (4.52)$$

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.

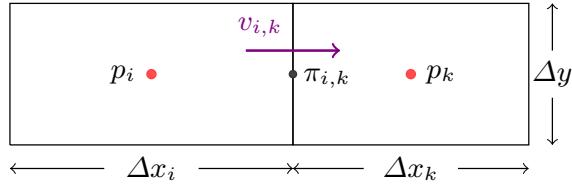


Fig. 4.11. Two cells used to derive the TPFA discretization for a 2D Cartesian grid

Example 4.1. 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 4.11. As above, we impose mass conservation inside each cell. For cell i this reads:

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

and likewise for cell k :

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

Next, we impose continuity of fluxes and face pressures,

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

which gives us two equations

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

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 [4], we showed how one could develop an efficient and self-contained MATLAB program that in approximately thirty compact lines solved the incompressible flow equation (4.46) 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 SPE 10 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 the author's webpage.

4.4.2 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 (4.46) 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 $F : \{1, \dots, n_c\} \rightarrow \{0, 1\}^{n_f}$ and maps a cell to the set of faces that constitute this cell. In a grid structure \mathbf{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, $C_1, C_2 : \{1, \dots, n_f\} \rightarrow \{1, \dots, n_c\}$. In a grid structure \mathbf{G} , C_1 is given by `G.faces.neighbors(:, 1)` and C_2 by `G.faces.neighbors(:, 2)`. This is illustrated in Figure 4.12.

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 $\mathbf{v} \in \mathbb{R}^{n_f}$. For a face f , the orientation of the flux $\mathbf{v}[f]$ is from $C_1(f)$ to $C_2(f)$. Hence, the total amount of matter leaving the cell c is given by

$$\text{div}(\mathbf{v})[c] = \sum_{f \in F(c)} \mathbf{v}[f] \mathbf{1}_{\{c=C_1(f)\}} - \sum_{f \in F(c)} \mathbf{v}[f] \mathbf{1}_{\{c=C_2(f)\}}. \quad (4.53)$$

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

$$\text{grad}(\mathbf{p})[f] = \mathbf{p}[C_2(f)] - \mathbf{p}[C_1(f)], \quad (4.54)$$

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

$$\int_{\Omega} p \nabla \cdot \vec{v} d\vec{x} + \int_{\Omega} \vec{v} \cdot \nabla p d\vec{x} = 0, \quad (4.55)$$

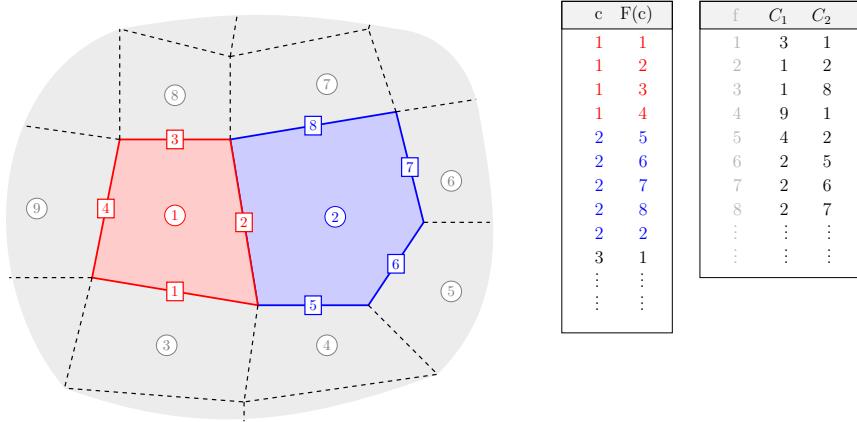


Fig. 4.12. Illustration of the mappings from cells to faces and from faces to cells used to define the discrete divergence and gradient operators.

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 $\mathbf{v} \in \mathbb{R}^{n_f}$ and $\mathbf{p} \in \mathbb{R}^{n_c}$, we have

$$\begin{aligned} \sum_{c \in S_c} \text{div}(\mathbf{v})[c] \mathbf{p}[c] &= \sum_{c \in S_c} \mathbf{p}[c] \left(\sum_{f \in F(c)} \mathbf{v}[f] \mathbf{1}_{\{c=C_1(f)\}} - \sum_{f \in F(c)} \mathbf{v}[f] \mathbf{1}_{\{c=C_2(f)\}} \right) \\ &= \sum_{c \in S_c} \sum_{f \in S_f} \mathbf{v}[f] \mathbf{p}[c] \mathbf{1}_{\{c=C_1(f)\}} \mathbf{1}_{\{f \in F(c)\}} \\ &\quad - \sum_{c \in S_c} \sum_{f \in S_f} \mathbf{v}[f] \mathbf{p}[c] \mathbf{1}_{\{c=C_2(f)\}} \mathbf{1}_{\{f \in F(c)\}} \end{aligned} \quad (4.56)$$

We can switch the order in the sums above and obtain

$$\begin{aligned} \sum_{c \in S_c} \sum_{f \in S_f} \mathbf{v}[f] \mathbf{p}[c] \mathbf{1}_{\{c=C_1(f)\}} \mathbf{1}_{\{f \in F(c)\}} &= \\ \sum_{f \in S_f} \sum_{c \in S_c} \mathbf{v}[f] \mathbf{p}[c] \mathbf{1}_{\{c=C_1(f)\}} \mathbf{1}_{\{f \in F(c)\}}. \end{aligned}$$

For a given face f , we have that $\mathbf{1}_{\{c=C_1(f)\}} \mathbf{1}_{\{f \in F(c)\}}$ is nonzero if and only if $c = C_1(f)$ and therefore

$$\sum_{f \in S_f} \sum_{c \in S_c} \mathbf{1}_{\{c=C_1(f)\}} \mathbf{1}_{\{f \in F(c)\}} \mathbf{v}[f] \mathbf{p}[c] = \sum_{f \in S_f} \mathbf{v}[f] \mathbf{p}[C_1(f)].$$

In the same way, we have

$$\sum_{c \in S_c} \sum_{f \in S_f} \mathbf{v}[f] \mathbf{p}[c] \mathbf{1}_{\{c=C_2(f)\}} \mathbf{1}_{\{f \in F(c)\}} = \sum_{f \in S_f} \mathbf{v}[f] \mathbf{p}[C_2(f)]$$

so that (4.56) yields

$$\sum_{c \in S_c} \operatorname{div}(\mathbf{v})[c] \mathbf{p}[c] + \sum_{f \in S_f} \operatorname{grad}(\mathbf{p})[f] \mathbf{v}[f] = 0. \quad (4.57)$$

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 C_1 and C_2 to $S_c \cup \{0\}$ so that, if a given face f satisfies $C_1(f) = 0$ or $C_2(f) = 0$ then it is external. Note that the `grad` operator only defines values on internal faces. Now taking external faces into account, we obtain

$$\begin{aligned} & \sum_{c \in S_c} \operatorname{div}(\mathbf{v})[c] \mathbf{p}[c] + \sum_{f \in S_f} \operatorname{grad}(\mathbf{p})[f] \mathbf{v}[f] \\ &= \sum_{f \in \bar{S}_f \setminus S_f} \left(\mathbf{p}[C_1(f)] \mathbf{1}_{\{C_2(f)=0\}} - \mathbf{p}[C_2(f)] \mathbf{1}_{\{C_1(f)=0\}} \right) \mathbf{v}[f], \end{aligned} \quad (4.58)$$

where \bar{S}_f denotes the set of internal and external faces. The identity (4.58) is the discrete counterpart to

$$\int_{\Omega} p \nabla \cdot \vec{v} d\vec{x} + \int_{\Omega} \vec{v} \cdot \nabla p d\vec{x} = \int_{\partial\Omega} p \vec{v} \cdot \vec{n} ds. \quad (4.59)$$

Going back to (4.46), we see that the vector $\mathbf{v} \in \mathbb{R}^{n_f}$ is a discrete approximation of the flux on faces. Given $f \in S_f$, we have

$$\mathbf{v}[f] \approx \int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f ds,$$

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 $\mathbf{p} \in \mathbb{R}^{n_c}$ and the discrete flux is given by the two-point flux approximation discussed in the previous section,

$$\mathbf{v}[f] = -\mathbf{T}[f] \operatorname{grad}(\mathbf{p})[f] \approx - \int_{\Gamma_f} K(x) \nabla p \cdot \vec{n}_f ds, \quad (4.60)$$

where $\mathbf{T}[f]$ denotes the transmissibility of the face f , as defined in (4.51). Hence, the discretization of (4.46) is

$$\operatorname{div}(\mathbf{v}) = q \quad (4.61a)$$

$$v = -T \operatorname{grad}(\mathbf{p}). \quad (4.61b)$$

where the multiplication in (4.61b) holds element-wise.

Example 4.2. To illustrate the use of the discrete operators, let us set up and solve the classical Poisson equation on a simple box geometry,

$$-\operatorname{div}(T \operatorname{grad}(\mathbf{p})) = q, \quad \Omega = [0, 1] \times [0, 1] \quad (4.62)$$

subject to no-flow boundary conditions with q consisting of a point source at $(0,0)$ and a point sink at $(1,1)$. First, we construct a small Cartesian grid

```
G = computeGeometry(cartGrid([5 5],[1 1]));
```

for which T equals a scalar multiple of the identity matrix and is therefore dropped for simplicity. The `div` and `grad` operators will be constructed as sparse matrices. To this end, we will use (4.54) and (4.57), which implies that the sparse matrix used to construct `div` is the negative transpose of the matrix that defines `grad`. Moreover, since we assume no-flow boundary conditions, we only need to let C_1 and C_2 account for internal connections:

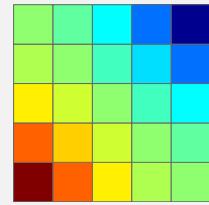
```
C = G.faces.neighbors;
C = C(all(C ~= 0, 2), :);
nf = size(C,1);
nc = G.cells.num;
D = sparse([(1:nf)'; (1:nf)'], C, ...
    ones(nf,1)*[-1 1], nf, nc);
grad = @x D*x;
div = @x -D'*x;
```

$$D = \begin{bmatrix} & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix} \quad \begin{array}{l} \frac{\partial}{\partial x} \\ \parallel \\ \frac{\partial}{\partial y} \end{array}$$

Once we have the discrete operators, we can write (4.62) in residual form, $\mathbf{f}(\mathbf{p}) = \mathbf{A}\mathbf{p} + \mathbf{q} = 0$, and then use automatic differentiation as discussed in Example A.3 on page 613 to obtain \mathbf{A} by computing $\partial\mathbf{f}/\partial\mathbf{p}$

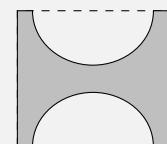
```
p = initVariablesADI(zeros(nc,1));
q = zeros(nc, 1); % source term
q(1) = 1; q(nc) = -1; % -> quarter five-spot

eq = div(grad(p))+q; % equation
eq(1) = eq(1) + p(1); % make solution unique
p = -eq.jac{1}\eq.val; % solve equation
```



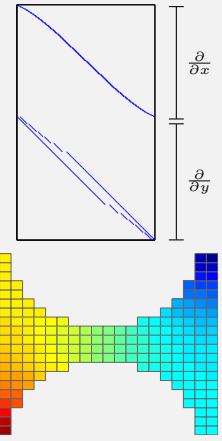
Next, we try to solve the same type of flow problem on a non-rectangular domain. That is, we still consider the unit square, but remove two half-circles of radius 0.4 centered at (0.5,0) and (0.5,1), respectively. To construct the corresponding grid, we use the fictitious grid approach from Section 3.1 (see Exercise 6 on page 67):

```
G = cartGrid([20 20],[1 1]);
G = computeGeometry(G);
r1 = sum(bsxfun(@minus,G.cells.centroids,[0.5 1]).^2,2);
r2 = sum(bsxfun(@minus,G.cells.centroids,[0.5 0]).^2,2);
G = extractSubgrid(G, (r1>0.16) & (r2>0.16));
```



The construction of the discrete operators is agnostic to the exact layout of the grid, and since the transmissibility matrix T is still a multiple of the identity matrix, since the grid cells are equidistant squares, we can simply reuse the exact same set-up as above:

```
% Grid information
C = G.faces.neighbors;
:
% Operators
D = sparse([(1:nf)'; (1:nf)'], C, ...
    ones(nf,1)*[-1 1], nf, nc);
:
% Assemble and solve equations
p = initVariablesADI(zeros(nc,1));
q = zeros(nc, 1);
q(1) = 1; q(nc) = -1;
eq = div(grad(p))+q;
eq(1) = eq(1) + p(1);
p = -eq.jac{1}\eq.val;
plotCellData(G,p);
```

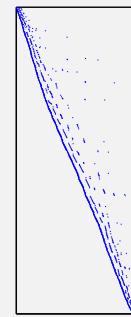


Notice that the D matrix has almost the same sparsity structure as in our first problem, except that the nonzero band now are curved since the number of cells in each column/row of the grid changes throughout the domain.

Example 4.3. To illustrate the power of the combination of an unstructured grid format and discrete differential operators, we also go through how you can use this technique to solve the Poisson equation on an unstructured grid. As an example, we use the Voronoi grid generated from the `seamount` data set shown in Figure 3.11 on page 74. Now comes the important point: *Because the discrete differential operators are defined in terms of the two general matrices C_1 and C_2 that describe the internal connections in the grid, their construction remains exactly the same as for the Cartesian grid:*

```
load seamount
G = pebi(triangleGrid([x(:) y(:)], delaunay(x,y)));
G = computeGeometry(G);

C = G.faces.neighbors;
C = C(all(C ~= 0, 2), :);
nf = size(C,1);
nc = G.cells.num;
D = sparse([(1:nf)'; (1:nf)'], C, ...
    ones(nf,1)*[-1 1], nf, nc);
grad = @x) D*x;
```



Here, the directional derivatives do not follow the axial directions and hence D will have a general sparse structure and not the banded structure we saw for the Cartesian grids in the previous example. Likewise, because the cell centers are no longer equidistant points on a uniform mesh, the diagonal entries in the transmissibility matrix will not be the same constant for all cells and hence cannot be scaled out of the discrete system. For historical reasons, MRST only supplies a routine for computing half-transmissibilities

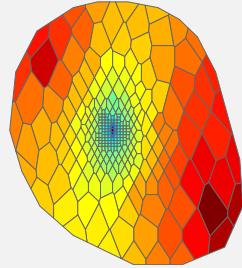
defined in (4.50) on page 138. These are defined for *all* faces in the grid. Since we have assumed no-flow boundary conditions, we hence only need to find the half-transmissibilities associated with the *interior* faces and compute their harmonic average to get the transmissibilities defined in (4.51):

```

hT = computeTrans(G, struct('perm', ones(nc,1)));
cf = G.cells.faces(:,1);
T = 1 ./ accumarray(cf, 1 ./ hT, [G.faces.num, 1]);
T = T(all(C~=0,2),:);

p = initVariablesADI(zeros(nc,1));
q = zeros(nc, 1); q([135 282 17]) = [-1 .5 .5];
eq = div(T.*grad(p))+q;
eq(1) = eq(1) + p(1);
p = -eq.jac{1}\eq.val;

```



You may also notice that we have changed our source terms slightly so that there is now a fluid sink at the center and fluid sources to the north-west and south-east. We will return to a more detailed discussion of the computation of transmissibilities and assembly of discrete linear systems in Section 5.2

4.4.3 Time-of-flight and tracer

The transport equations (4.39) and (4.41) can be written on the common form

$$\nabla \cdot (u\vec{v}) = h(\vec{x}, u), \quad (4.63)$$

where $u = \tau$ and $h = \phi + \tau \nabla \cdot \vec{v}$ for time-of-flight and $u = C$ and $h = q_C + C \nabla \cdot \vec{v}$ for the artificial tracer.

To discretize the steady transport equation (4.63), we integrate it over a single grid cell Ω_i and use Gauss' divergence theorem to obtain

$$\int_{\partial\Omega_i} u \vec{v} \cdot \vec{n} ds = \int_{\Omega_i} h(\vec{x}, u(\vec{x})) d\vec{x}.$$

In Section 4.4.1 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

$$\int_{\Gamma_{ik}} u \vec{v} \cdot \vec{n} ds = u_{ik} v_{ik}. \quad (4.64)$$

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 propagating in the transport equation, we can instead use the so-called upwind value

$$u_{ik} = \begin{cases} u_i, & \text{if } v_{ik} \geq 0, \\ u_k, & \text{otherwise.} \end{cases} \quad (4.65)$$

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 $\mathbf{u} \in \mathbb{R}^{n_c}$ and h by a vector function $\mathbf{h}(\mathbf{u}) \in \mathbb{R}^{n_c}$, the transport equation (4.63) can be written in the discrete form

$$\operatorname{div}(\mathbf{u}\mathbf{v}) = \mathbf{h}(\mathbf{u}). \quad (4.66)$$

We also substitute the expression for \mathbf{v} from (4.61b) and use (4.65) to define u at each face f . Then, we define, for each face $f \in S_f$,

$$(\mathbf{u}\mathbf{v})[f] = \mathbf{u}^{uw}[f] \mathbf{T}[f] \operatorname{grad}(\mathbf{p})[f], \quad (4.67)$$

where

$$\mathbf{u}^{uw}[f] = \begin{cases} \mathbf{u}[C_1(f)], & \text{if } \operatorname{grad}(\mathbf{p})[f] > 0, \\ \mathbf{u}[C_2(f)], & \text{otherwise.} \end{cases} \quad (4.68)$$

Time-of-flight and tracer distributions can of course also be computed based on tracing streamlines by solving the ordinary differential equations (4.35). The most commonly used method for tracing streamlines on hexahedral grids is a semi-analytical tracing algorithm introduced by Pollock [216], 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 [64, 219, 121, 170, 97, 169, 137]. 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.

COMPUTER EXERCISES:

22. Compare the discrete differentiation operators for selected grids from Chapter 3, e.g., Exercises 6 to 8 on page 68 and Exercises 11 and 12 on page 79. Can you explain the differences?

Incompressible Solvers

A simulation model can be considered to consist of two main parts; the first part describes the reservoir rock and the second part describes the mathematical laws that govern fluid behavior. We have already discussed how to model the reservoir rock and its petrophysical properties in Chapters 2 and 3 and shown how the resulting geological models are represented in MRST using a grid object, usually called `G`, that describes the geometry of the reservoir, and a rock object, usually called `rock`, that describes petrophysical parameters. From the properties in these objects, one can compute spatial discretization operators that are generic and not tied to a particular set of flow equations as discussed in Section 4.4.2.

To form a full simulation model for porous media flow, however, we also need to introduce forcing terms and fluid properties. In MRST, the fluid behavior is represented as a fluid object that describes basic fluid properties such as density, viscosity, and compressibility. These fluid objects can then be extended to model more complex behavior for specific models, for instance to include properties like relative permeability and capillary pressure that describe the interaction between a multiphase flow interacts and a porous rock. Forcing terms other than gravity are represented similarly using objects with data structures that are specific to boundary conditions, (volumetric) source terms, and models of injection and production wells. In addition, it is convenient to introduce a state object holding the reservoir states (primary unknowns and derived quantities) like pressure, fluxes, face pressures, etc.

There are two different ways the data objects outlined above can be combined to form a full simulator. In Example 4.2, we saw how one can use discrete differential operators to write the flow equations in residual form and then employ automatic differentiation to linearize and form a linear system. Whereas this technique is elegant and will prove highly versatile for compressible flow models later in the book, it is an overkill for incompressible single-phase flow, since these equations already are linear. In this chapter we therefore outline how one can use a classic procedural approach to implement the discretized flow equations from the previous chapter. We start by outlining

ing the data structures and constructors needed to set up fluid properties and forcing terms, and once this is done, we move on to discuss in detail how to build two-point discretizations and assemble and solve corresponding linear systems. The codes presented are simplified version of the basic flow solvers for incompressible flow that are implemented in the add-on modules `incomp` and `diagnostics` of MRST. Then at the end of the chapter, we go through a few examples and give all code lines that are necessary for full simulation setups with various driving mechanisms.

5.1 Basic data structures in a simulation model

In the previous chapter, we showed an example of a simple flow solver that did not contain any fluid properties and assumed no-flow boundary conditions and point sources as the only forcing term. In this section we will outline the basic data structures that are needed to set up more comprehensive single-phase simulation models.

5.1.1 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 5.1.2) 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.

5.1.2 Reservoir states

To hold the dynamic state of the reservoir, MRST uses a special data 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 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.

5.1.3 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 benefit 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 incompressibility.

5.1.4 Boundary conditions

As discussed in Section 4.3.1, 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 fluxes, 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 5.1.3.

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 incompressibility.

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 must 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 cumulative flux for all faces that make up the global side to be equal the specified value.

5.1.5 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 4.3.2, this relation can be written for each perforation as

$$v_p = J(p_i - p_f) \quad (5.1)$$

where J 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 the 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. By convention, the structure used to represent wells in MRST is called `w`, and 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 6
- **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 **isempty(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**.

5.2 Incompressible two-point pressure solver

The two-point flux-approximation scheme introduced in Section 4.4.1 is implemented as two different routines in the `incomp` module:

```
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 5.1.3–5.1.5. Notice that `computeTrans` may fail to compute sensible transmissibilities if 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 5.4, which contains 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 `incomp`. Instead, we 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 which has been created for pedagogical purposes. The standard `computeTrans` function from `mrst-core` 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 2.4.3 and 2.5.5. Likewise, the `incompTPFA` solver from the `incomp` module 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.

Assume that we have a standard grid `G` that contains cell and face centroids, e.g., as computed by the `computeGeometry` function discussed in Section 3.4. Then, the essential code lines of `simpleComputeTrans` are as follows: First, we define the vectors $\vec{c}_{i,k}$ from cell centroids to face centroids, see Figure 5.1. 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.

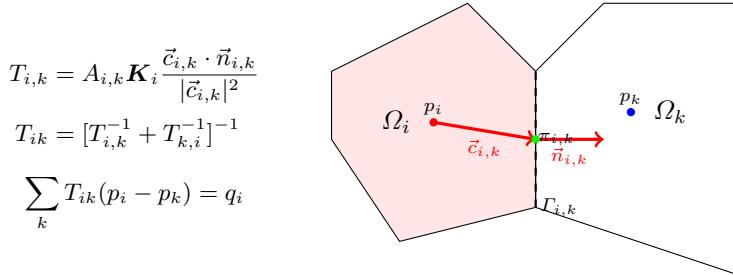


Fig. 5.1. Two-point discretization on general polyhedral cells

```
hf = G.cells.faces(:,1);
hf2cn = gridCellNo(G);
C = G.faces.centroids(hf,:) - 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 (4.50). 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(hf, 1)) - 1;
N = bsxfun(@times, sgn, G.faces.normals(hf,:));
```

The permeability tensor may be stored in different formats, as discussed in Section 2.5, 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 is 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 that 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(hf, 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 very powerful and is used a lot in MRST in place of nested for-loops. In fact, we will employ 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);
n1 = G.faces.neighbors(i,1);
n2 = G.faces.neighbors(i,2);
d = accumarray([n1; n2], 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 = [n1; n2; (1:nc)'];
J = [n2; n1; (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 fix the pressure in the first cell to zero, before solving the system:

```
A(1) = 2*A(1);
p = mldivide(A, rhs);
```

To solve the system, we rely on MATLAB's default solver `mldivide`, which uses a complex flow chart to check the structure of the matrix to see whether it is square, diagonal, (permuted) triangular, tridiagonal, banded, or Hermitian and choose a specialized and efficient solver accordingly. By running the command `spparms('spumoni', 2)` before you call `mldivide`, you can tell MATLAB to output information about which linear solver it chooses and the tests leading up the specific choice. For the type of sparse matrices we consider here, the end result is a call to a direct solver from UMFPACK implementing unsymmetric, sparse, multifrontal LU factorization [76, 75]. While this solver is

efficient for small to medium-sized systems, larger systems are more efficiently solved using more problem-specific solvers like a (preconditioned) multilevel method. To provide flexibility, the linear solver can be passed as a function-pointer argument to both `incompTPFA` and `simpleIncompTPFA`. (One example of a solver that integrates well with MRST is AGMG solver [203, 10], which implements an aggregation-based algebraic multigrid method and is available at no cost for academic research and teaching.)

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 general Dirichlet or Neumann 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.

We will shortly discuss several examples that demonstrate how this code can be used to solve flow problems on structured and unstructured grids. However, before doing so, we outline another flow solver from the `diagnostics` module visualizing flow patterns

5.3 Upwind solver for time-of-flight and tracer

The `diagnostics` module provides various functionality that can be used to probe a reservoir model to establish communication patterns between inflow and outflow regions, time lines for fluid movement, and various measures of reservoir heterogeneity. At the heart of this module, lies the function

```
tof = computeTimeOfFlight(state, G, rock, mech1, obj1, ...)
```

which implements the upwind, finite-volume discretization introduced in Section 4.4.3 for solving the time-of-flight equation $\vec{v} \cdot \nabla \tau = \phi$ to compute the time it takes a neutral particle to travel from the nearest fluid source or inflow boundary to each point in the reservoir. Here, the `mech` arguments specify the drive mechanism ('src', 'bc', and/or 'wells') specified in terms of specific objects `obj`, as discussed in Sections 5.1.3 to 5.1.5. The same routine can also compute tracer concentrations that can be used to define volumetric partitions if the user specifies extra input parameters. Likewise, the backward time-of-flight, i.e., the time it takes to travel from any point in the reservoir and to the nearest fluid sink or outflow boundary, can be computed from the same

equation if we change the sign of the flow field and modify the boundary conditions and/or source terms accordingly. 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, which may be described as source/sink terms in `src` and/or as wells in `W`,

```
[qi,qs] = deal([]);
if ~isempty(W),
    qi = [qi; vertcat(W.cells)];
    qs = [qs; vertcat(state.wellSol.flux)];
end
if ~isempty(src),
    qi = [qi; src.cell];
    qs = [qs; src.rate];
end
```

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 can start building the upwind flux discretization matrix A . The off-diagonal entries are 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);
out = min(state.flux(i), 0);
in = max(state.flux(i), 0);
```

The diagonal entry equals the outflux minus the divergence of the velocity, which can be obtained by summing the off-diagonal rows. This will give the correct equation in all cell except for those with a positive fluid source. Here, the net outflux equals the divergence of the velocity and we hence end up with an undetermined equation. In these cells, we can as a resonable approximate¹ set the time-of-flight to be equal the time it takes to fill the cell, which means that the diagonal entry should be equal the fluid rate inside the cell.

```
n = double(G.faces.neighbors(i,:));
inflow = accumarray([n(:, 2); n(:, 1)], [in; -out]);
d = inflow + max(q+qb, 0);
```

Having obtained diagonal and all the nonzero off-diagonal elements, we can assemble the full matrix

```
nc = G.cells.num;
A = sparse(n(:,2), n(:,1), in, nc, nc)...
+ sparse(n(:,1), n(:,2), -out, nc, nc);
A = -A + spdiags(d, 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 [187, 186]. 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 back-substitution 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. However, for models of real petroleum assets described on stratigraphic grids (see Chapter 3.3), it

¹ Notice, however, that to get the correct values for 1D cases, it is more natural to set time-of-flight equal *half* the time it takes to fill the cell.

is often necessary to preprocess flux fields to get rid of numerical clutter that would otherwise introduce large irreducible blocks inside stagnant regions. By specifying optional parameters to `computeTimeOfFlight`, the function will get rid of such small cycles in the flux field and set the time-of-flight to a prescribed upper value in all cells that have sufficiently small influx. This tends to reduce the computational cost significantly for large models with complex geology and/or significant compressibility effects.

In addition to time-of-flight, we can compute stationary tracers as discussed in Section 4.3.4. 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 gives the indexes of the cells that emit a unique tracer. 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 , the rate on the right-hand side must also be doubled.

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 the number of right-hand columns MATLAB can be handled in one solve. We will return to a more thorough discussion of the tracer partitions in the next chapter and show how these can be used to delineate connectivities within the reservoir. In the rest of this chapter, we will consider time-of-flight and streamlines as a means to study flow patterns in reservoir models.

5.4 Simulation examples

You have now been introduced to all the functionality from the `incomp` module that is necessary to solve a single-phase flow problem as well as the time-of-flight solver from the `diagnostics` module, which can be used to compute time lines in the reservoir. In following, we will discuss several examples, in which we demonstrate step-by-step how to set up a flow model, solve it, and visualize and analyze the resulting flow field. Complete codes can be found in the `1phase` directory of the `mrst-book` module.

5.4.1 Quarter five-spot

As our first example, we show how to solve $-\nabla \cdot (\mathbf{K} \nabla p) = q$ with no-flow boundary conditions and two source terms at diagonally opposite corners of a 2D Cartesian grid covering a 500×500 m² area. This setup mimics a standard quarter five-spot well pattern, which you already have encountered in Figure 4.7 on page 131 when we discussed well models. 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 = makeRock(G, 100*milli*darcy, .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 flow, we use a fluid source in the southwest corner and a fluid sink in the northeast corner. The time scale of the problem is defined by the strength of the source term. Here, we set the source terms so that a unit time corresponds to the injection of one pore volume of fluids. By convention, 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:

```
src =
cell: [2x1 double]
rate: [2x1 double]
sat: []
```

The `src.cell` gives the cell numbers where the source term is nonzero, and the vector `src.rate` specifies the fluid rates, which by convention are positive for inflow into the reservoir and negative for outflow from the reservoir. The last data element `src.sat` specifies fluid saturations, which only has meaning for multiphase flow models and hence is set to be empty here.

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 driving forces to the flow solver, which 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 inter-cell 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 5.2 shows the resulting pressure distribution. To improve the visualization of the flow field, we show streamlines. In MRST, Pollock's method [216] for semi-analytical tracing of streamlines has been implemented in the `streamlines` add-on module. Here, we will use this functionality to trace streamlines forward and backward, starting from the midpoint of all cells along the NW–SE diagonal in the grid

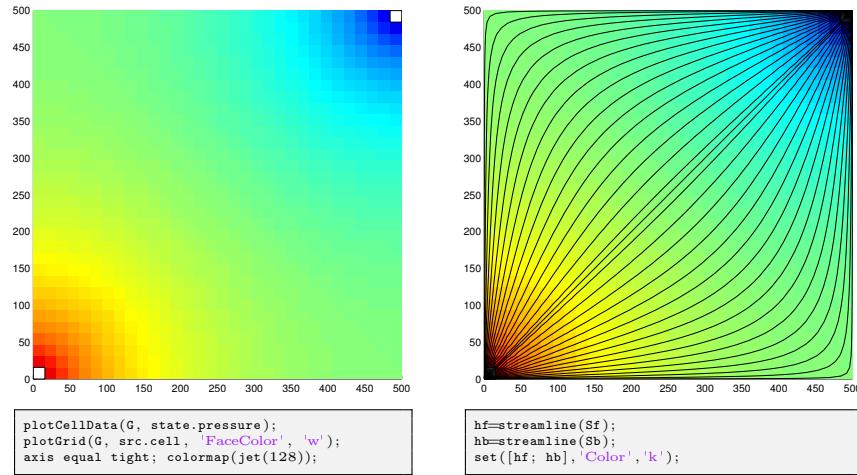


Fig. 5.2. Solution of the quarter five-spot problem on a 32×32 uniform grid. The left plot shows the pressure distribution and in the right plot we have imposed streamlines passing through centers of the cells on the NW–SE diagonal.

```

mrstModule add streamlines;
seed = (nx:nx-1:ny).';
Sf = pollock(G, state, seed, 'substeps', 1);
Sb = pollock(G, state, seed, 'substeps', 1, 'reverse', true);

```

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 5.2.

To get a better picture of how fast the fluids will flow through our domain, we solve the time-of-flight equation (4.39) 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 5.3, 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

```

tobf = 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 5.3 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 gives the total time it takes a fluid particle to travel through the reservoir, from an inflow point to an outflow

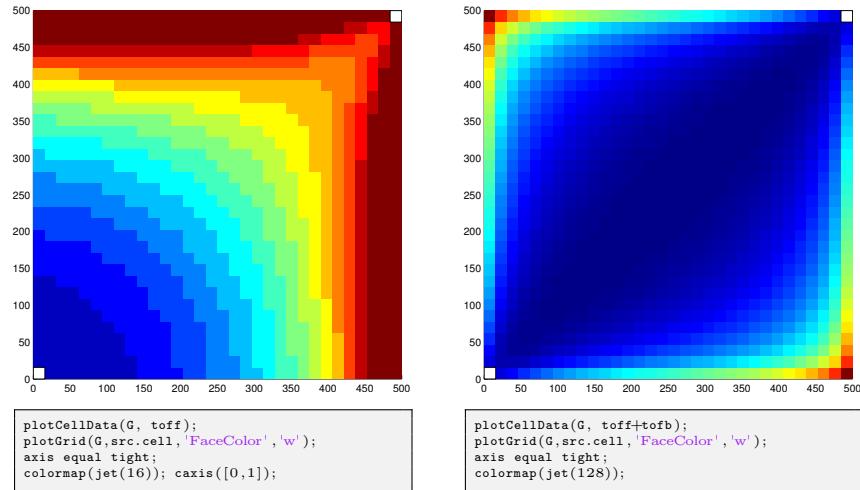


Fig. 5.3. Solution of the quarter five-spot problem on a 32×32 uniform grid. The left plot shows time-of-flight plotted with a few color levels to create a crude contouring effect. The right plot shows a plot of the total travel time to distinguish high-flow and stagnant regions.

point. The total travel time can be used to visualize high-flow and stagnant regions as demonstrated in the right plot of Figure 5.3.

COMPUTER EXERCISES:

23. Run the quarter five-spot example with the following modifications:
 - a) 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.
 - b) Replace the Cartesian grid by the locally refined grid from Exercise 11 on page 79
 - c) Replace the homogeneous permeability by a heterogeneous permeability derived from the Carman–Kozeny relation (2.6)
 - d) Set the domain to be a single layer of the SPE 10 model. Hint: use `getSPE10rock()` to sample the petrophysical parameters and remember to convert to SI units.
- Notice that the `pollock` function may not work for non-Cartesian grids.
24. Construct a grid similar to the one in Exercise 6 on page 67, except that the domain is given a 90° flip so that axis of the cylindrical cut-outs align with the z -direction. Modify the code presented above so that you can compute a *five-spot* setup with one injector near each corner and a producer in the narrow middle section between the cylindrical cut-outs.

5.4.2 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 $1 \times 1 \text{ km}^2$ 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 3.34):

```
[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(makeModel13([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 m³/day on the east boundary and a Dirichlet condition with fixed pressure of 50 bar on the west boundary:

```
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 5.4. 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

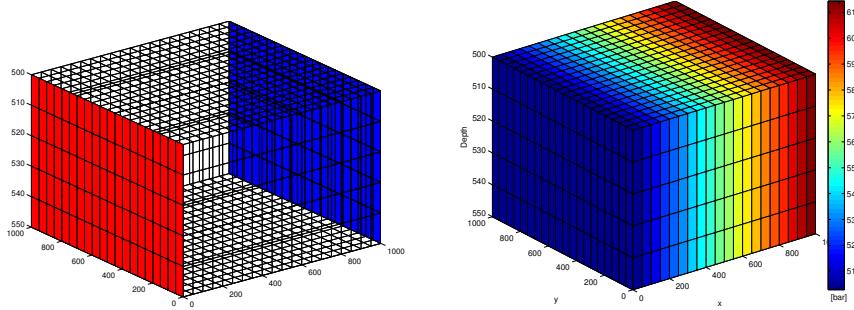


Fig. 5.4. First example of a flow driven by boundary conditions. In the left plot, faces with Neumann conditions are marked in blue and faces with Dirichlet conditions are marked in red. The right plot shows the resulting pressure distribution.

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,

```
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);
```

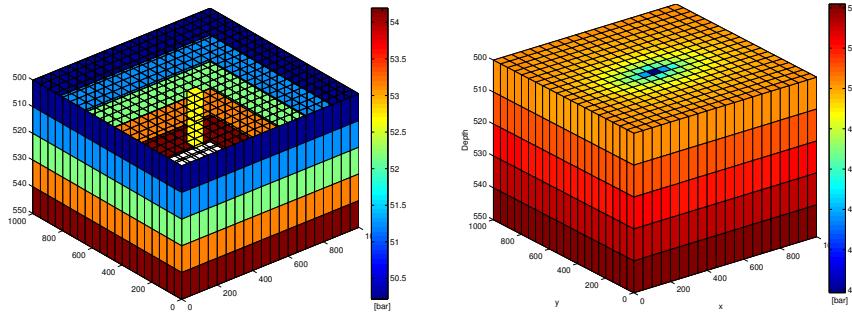


Fig. 5.5. A reservoir with hydrostatic boundary condition and fluid extracted from a sink penetrating two cells in the upper two layers of the model. The left plot shows the boundary and the fluid sink, while the right plot shows the resulting pressure distribution.

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 5.5 and the resulting pressure distribution to the right. The fluid sink will cause a pressure drawdown, which will have an ellipsoidal shape because of the anisotropy in the permeability field.

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 and 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 5.6, where we have tried to specify boundary conditions using the same procedure as in the linear pressure-drop example (Figure 5.4).

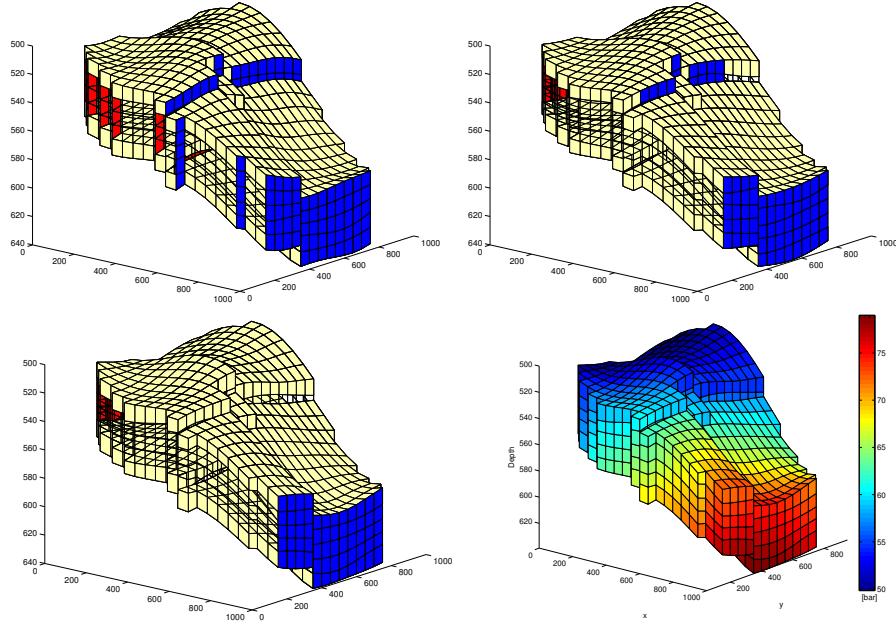


Fig. 5.6. Specifying boundary conditions along the outer perimeter of a corner-point model. The upper-left plot shows the use of `fluxside` (blue color) and `pside` (red color) to set boundary conditions on the east and west global boundaries. In the upper-right point, the same functions have been used 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.

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 5.6. 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. On these faces, we distribute the total flux to individual faces according to the face area. 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)));

```

The Dirichlet condition can be specified in a similar manner.

COMPUTER EXERCISES:

25. 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.
26. Apply the production setup from Figure 5.5 on page 167, with hydrostatic boundary conditions and fluids extracted from two cells at the midpoint of the model, to the model depicted in Figure 5.6 on the facing page.
27. Compute the flow patterns for all the bed models in data sets `BedModels1` and `BedModel2` subject to linear pressure drop first in the x and then in the y -direction. These data sets contain models of small-scale heterogeneity and that can be used to compute representative properties in simulation models on a larger scale. A linear pressure drop is the most wide-spread computational setup used for flow-based upscaling. What happens if you try to specify flux conditions?
28. Consider models from the `CaseB4` data set. Use appropriate boundary conditions to drive flow across the faults and compare flow patterns computed on the pillar and on the stair-stepped grid, as well as solutions computed for the two different model resolutions. Can you explain any differences you observe?

5.4.3 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 $1 \times 1 \text{ km}^2$ 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 93 to generate a set of points inside $[-1, 1] \times [-1, 1]$ that are graded radially towards the origin (see e.g., Figure 3.25),

```
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 = makeRock(g{i}, 100*milli*darcy, 0.2);
    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} = incomTPFA(initResSol(g{i},0,1), ...
        g{i}, hT, fluid, 'src', src{i}, 'bc', bc{i}, 'MatrixOutput', true);

    [t0f{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 5.7, while Figure 5.8 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 each cell and its four neighbors in the cardinal directions. Even though this discretization is not able to predict the complete drawdown at the center (the reference solution predicts a pressure slightly below 40 bar), it captures the shape of the drawdown 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 $r \rightarrow 300$ 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 drawdown 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 5.8, 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 5.7 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 drawdown in the near-well region. But to be fair, the grid was not generated for this purpose either.

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,

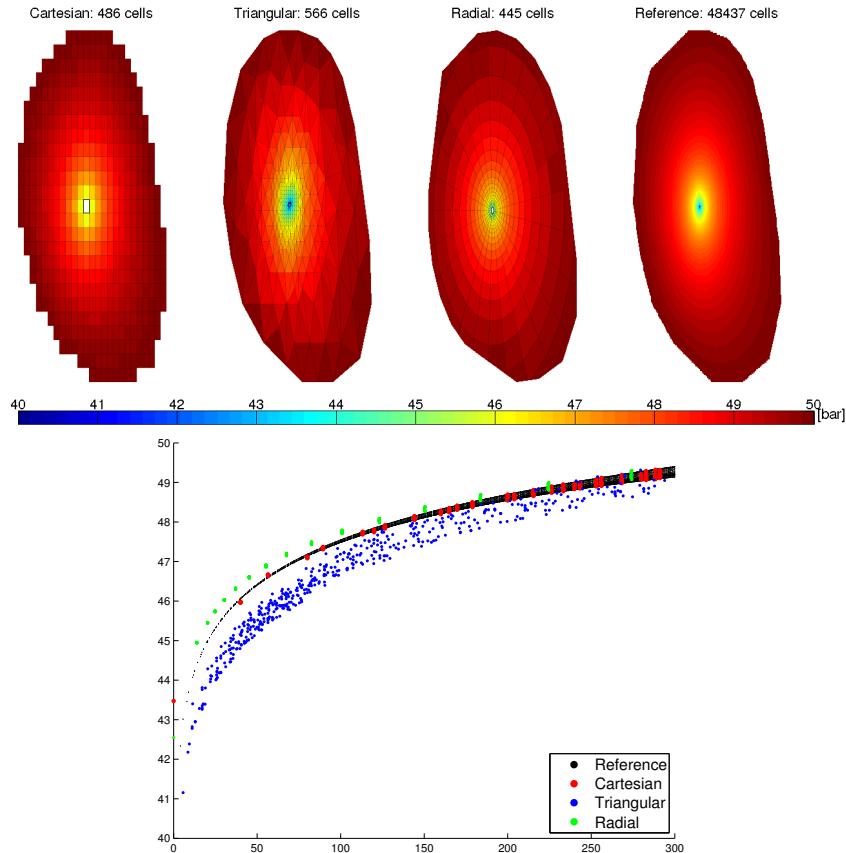


Fig. 5.7. Comparison of the pressure solution for three different grid types: uniform Cartesian, triangular, and a graded radial grid. The scattered points used to generate the triangular domain and limit the reservoir are sampled from the `seamount` data set and scaled to cover a $1 \times 1 \text{ km}^2$ area. Fluids are drained from the center of the domain, assuming a constant pressure of 50 bar at the perimeter.

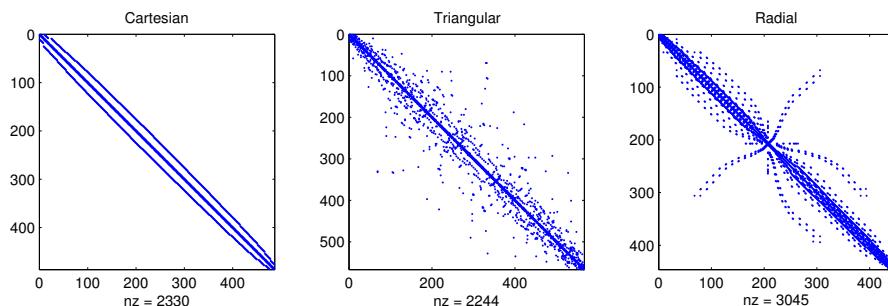


Fig. 5.8. Sparsity patterns for the TPFA stencils on the three different grid types shown in Figure 5.7.

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, due to the inherent difference in the discretizations.

In Figure 5.9, 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 `dmpem` to compute a Dulmage–Mendelsohn decomposition. As pointed out in Section 5.3, 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:

29. Compare the sparsity patterns resulting from the potential ordering and use of `dmpem` for both the upwind and the TPFA matrices.
30. Investigate the flow patterns in more details using forward time-of-flight, travel time, and streamlines.
31. Replace the boundary conditions by a constant influx, or set pressure values sampled from a radially symmetric pressure solution in an infinite domain.

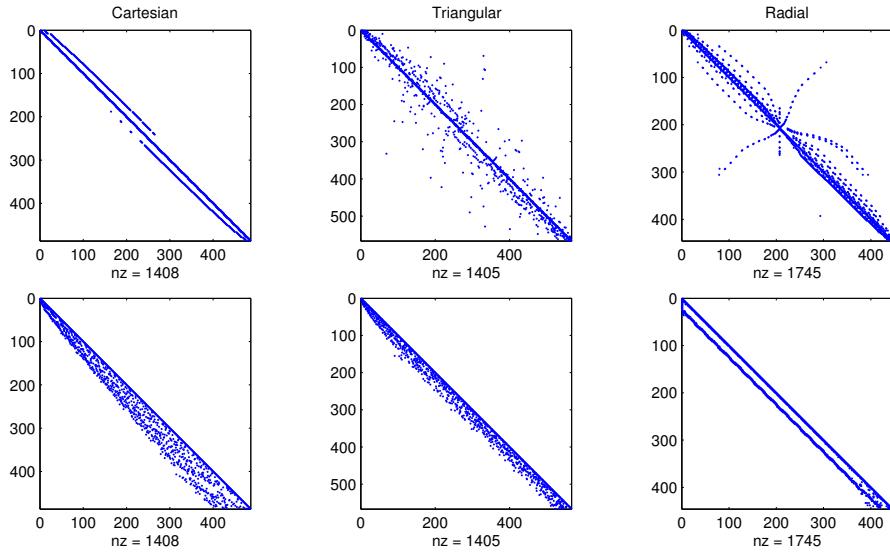


Fig. 5.9. Sparsity patterns for the upwind stencils used to compute time-of-flight on the three different grid types shown in Figure 5.7. In the lower row, the matrices have been permuted to lower-triangular form by sorting the cell pressures in ascending order.

5.4.4 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 4.3.2, 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 (4.34). 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 5.1.5. 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 2.5.5.

Box reservoir

We consider a reservoir consisting of a homogeneous $500 \times 500 \times 25 \text{ m}^3$ sand box with a isotropic permeability of 100 mD, represented on a regular $20 \times 20 \times 5$ 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 = makeRock(G, 100*milli*darcy, .2);
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 m³ 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., 10⁵ 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: cells: [5x1 double] type: 'rate' val: 0.0347 r: 0.1000 dir: [5x1 char] WI: [5x1 double] dZ: [5x1 double] name: 'I' compi: 1 refDepth: 0 sign: 1	Well #2: cells: [20x1 double] type: 'bhp' val: 100000 r: 0.1000 dir: [20x1 char] WI: [20x1 double] dZ: [20x1 double] name: 'P' compi: 1 refDepth: 0 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 5.10. 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)));
```

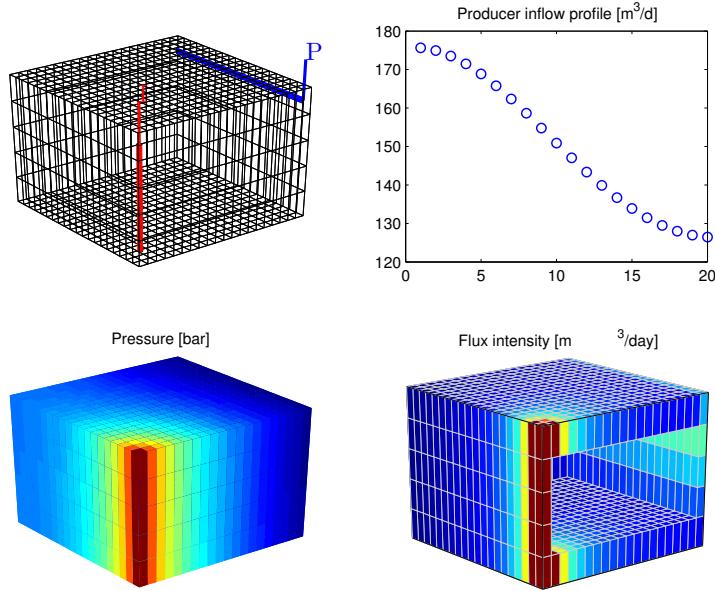


Fig. 5.10. Solution of a single-phase, incompressible flow problem inside a box reservoir with a vertical injector and a horizontal producer.

Shallow-marine reservoir

In the final example, we will return to the SAIGUP model discussed in Section 3.5. 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 3.5, 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));
```

Similar safeguards are implemented in most commercial simulators.

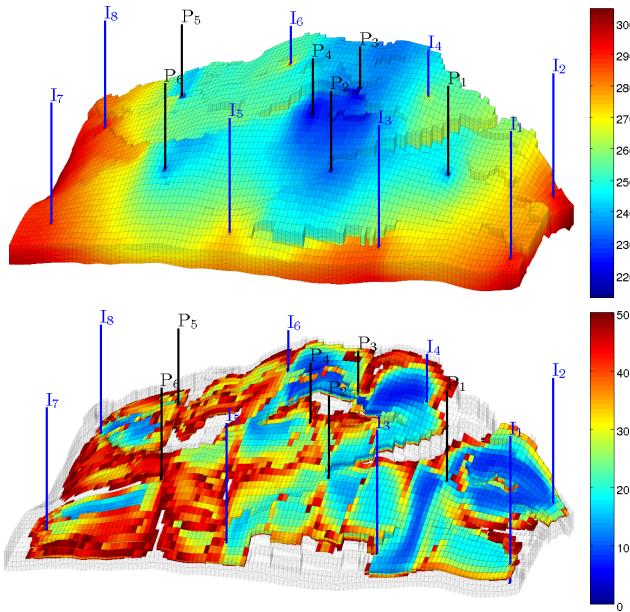


Fig. 5.11. Incompressible, single-phase simulation of the SAIGUP model. The upper plot shows pressure distribution, and the lower plot shows cells with total travel time less than fifty years.

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 5.11 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 draw-

down 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 5.4.1 (see Figure 5.3), we saw that this was not the case, and one cannot expect that this will happen here, either. The lower plot in Figure 5.11 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, one 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. We will come back to a more detailed discussion of flow patterns and volumetric connections in Section 13.4.2.

COMPUTER EXERCISES:

32. 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?
33. 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 perforate (complete) it in five cells?
34. 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?
35. Use the function `getSPE10setup` to set up an incompressible, single-phase version of the full SPE 10 benchmark. Compute pressure, time-of-flight and tracer concentrations associated with each well. Hint: You may need to replace MATLAB's standard backslash-solver by a highly-efficient iterative solver like AGMG [203, 10] to get reasonable computational performance. Also, beware that you may run out of memory if your computer is not sufficiently powerful.

6

Consistent Discretizations on Polyhedral Grids

In the previous chapters, we have used the two-point flux-approximation (TPFA) scheme introduced in Section 4.4.1 to discretize the elliptic Laplace operator $\mathcal{L} = \nabla \cdot \mathbf{K} \nabla$ 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

$$v_{ik} = T_{ik}(p_i - p_k), \quad (6.1)$$

where the transmissibility T_{ik} depends on the geometry of the two cells and the associated permeability tensors \mathbf{K}_i and \mathbf{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 \mathbf{K} and hence will not generally be convergent. To see this, we assume that \mathbf{K} is a homogeneous symmetric tensor

$$\mathbf{K} = \begin{bmatrix} K_{xx} & K_{xy} \\ K_{xy} & K_{yy} \end{bmatrix}.$$

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 6.1. Using Darcy's law gives that $\vec{v} \cdot \vec{n} = -(K_{xx}\partial_x p + K_{xy}\partial_y p)$, from which it follows that the flux across the interface will have two components, one orthogonal and one transverse,

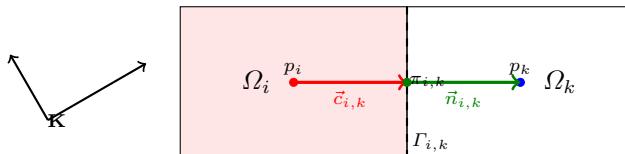


Fig. 6.1. Two cells in a Cartesian grid and a full permeability tensor whose principal axes are not aligned with the coordinate system.

$$v_{ik} = - \int_{\Gamma_{ik}} \mathbf{K} \nabla p \cdot \vec{n} \, ds = - \int_{\Gamma_{ik}} (\mathbf{K}_{xx} \partial_x p + \mathbf{K}_{xy} \partial_y p) \, ds$$

that correspond to the x -derivative and the y -derivative of the pressure, respectively. In the two-point approximation (6.1), 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 $\mathbf{K}_{xy} \partial_y p$ and will hence generally not be consistent. The only exception is if $\mathbf{K}_{xy} \equiv 0$ 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 (4.50) of the two-point scheme, two write

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

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

$$\vec{n}_{i,j} \cdot \mathbf{K} \vec{n}_{i,k} = 0, \quad \forall j \neq k, \quad (6.2)$$

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 4.10. 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

$$v_{i,k} = - \int_{\Gamma_{ik}} (p_1 \mathbf{K} \vec{c} \cdot \vec{n} + p_2 \mathbf{K} \vec{c}_\perp \cdot \vec{n}) \, ds.$$

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

$$0 = \mathbf{K} \vec{c}_\perp \cdot \vec{n} = (K_1 c_2, -K_2 c_1) \cdot (n_1, n_2) = (K_1 n_1 c_2 - K_2 n_2 c_1) = (\mathbf{K} \vec{n}) \times \vec{c}.$$

In other words, a sufficient condition for a polygonal grid to be K -orthogonal is that $(\mathbf{K} \vec{n}_{i,k}) \parallel \vec{c}_{i,k}$ 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

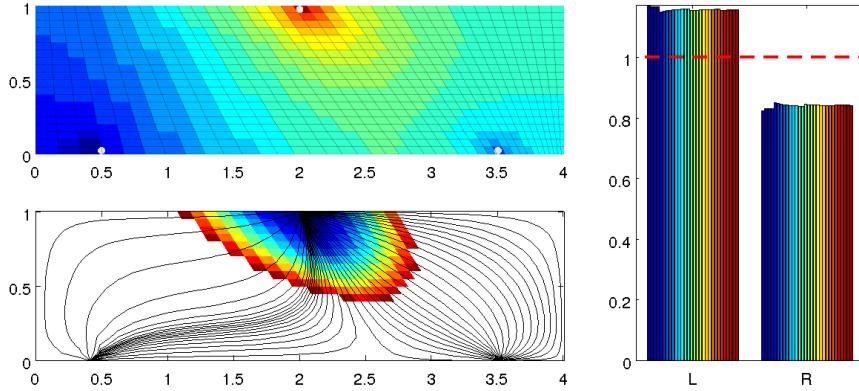


Fig. 6.2. Solution of a symmetric flow problem in a homogeneous domain using the TPFA method on a skew grid that is not K -orthogonal. The upper-left plot shows the pressure distribution, whereas the lower-left plot shows streamlines and time-of-flight values less than 0.25 PVI. The plot to the right shows time-of-flight in left (L) and right (R) sink for a series of refined grids of dimension $(20n + 1) \times 10n$ for $n = 1, \dots, 30$. 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 5.7 in Section 5.4.3, where the use of a triangular grids caused large deviations from what should have been an almost symmetric pressure drawdown. 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 6.2 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.

6.1 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 [46], 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 6.2 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.

6.1.1 Continuous formulation

We start by restating the continuous flow equation in full detail:

$$\nabla \cdot \vec{v} = q, \quad \vec{v} = -\mathbf{K} \nabla p, \quad \vec{x} \in \Omega \subset \mathbb{R}^d \quad (6.3)$$

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

$$H_0^{\text{div}}(\Omega) = \{\vec{v} \in L^2(\Omega)^d : \nabla \cdot \vec{v} \in L^2(\Omega) \text{ and } \vec{v} \cdot \vec{n} \text{ on } \partial\Omega\}, \quad (6.4)$$

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 (6.3) now reads: find a pair (p, \vec{v}) that lies in $L^2(\Omega) \times H_0^{\text{div}}(\Omega)$ and satisfies

$$\begin{aligned} \int_{\Omega} \vec{u} \cdot \mathbf{K}^{-1} \vec{v} d\vec{x} - \int_{\Omega} p \nabla \cdot \vec{u} d\vec{x} &= 0, & \forall \vec{u} \in H_0^{\text{div}}(\Omega), \\ \int_{\Omega} w \nabla \cdot \vec{v} d\vec{x} &= \int_{\Omega} q w d\vec{x}, & \forall w \in L^2(\Omega). \end{aligned} \quad (6.5)$$

The first equation follows by multiplying Darcy's law by \mathbf{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 (6.5) can be written on a more compact form if we introduce the following three inner products

$$\begin{aligned} b(\cdot, \cdot) : H_0^{\text{div}} \times H_0^{\text{div}} &\rightarrow \mathbb{R}, & b(\vec{u}, \vec{v}) &= \int_{\Omega} \vec{u} \cdot \mathbf{K}^{-1} \vec{v} d\vec{x} \\ c(\cdot, \cdot) : H_0^{\text{div}} \times L^2 &\rightarrow \mathbb{R}, & c(\vec{u}, p) &= \int_{\Omega} p \nabla \cdot \vec{u} d\vec{x} \\ (\cdot, \cdot) : L^2 \times L^2 &\rightarrow \mathbb{R}, & (q, w) &= \int_{\Omega} qw d\vec{x}, \end{aligned} \quad (6.6)$$

and write

$$\begin{aligned} b(\vec{u}, \vec{v}) - c(\vec{u}, p) &= 0 \\ c(\vec{v}, w) &= (q, w) \end{aligned} \quad (6.7)$$

The mixed formulation (6.7) can alternatively be derived by minimizing the energy functional

$$I(\vec{v}) = \frac{1}{2} \int_{\Omega} \vec{v} \cdot \mathbf{K}^{-1} \vec{v} d\vec{x},$$

over all functions $\vec{v} \in H_0^{\text{div}}(\Omega)$ that are mass conservative, i.e., subject to the constraint

$$\nabla \cdot \vec{v} = q.$$

The common strategy for solving such minimization problems is to introduce a Lagrangian functional, which in our case reads

$$L(\vec{v}, p) = \int_{\Omega} \left(\frac{1}{2} \vec{v} \cdot \mathbf{K}^{-1} \vec{v} - p(\nabla \cdot \vec{v} - q) \right) d\vec{x} = \frac{1}{2} b(\vec{v}, \vec{v}) - c(\vec{v}, p) + (p, q),$$

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

$$0 = \frac{\partial L}{\partial \vec{v}} \vec{u} = \lim_{\varepsilon \rightarrow 0} \frac{1}{\varepsilon} [L(\vec{v} + \varepsilon \vec{u}, p) - L(\vec{v}, p)] = b(\vec{u}, \vec{v}) - c(\vec{u}, p)$$

must be zero for all $u \in H_0^{\text{div}}(\Omega)$. Similarly, we can show that

$$0 = \frac{\partial L}{\partial p} w = \lim_{\varepsilon \rightarrow 0} \frac{1}{\varepsilon} [L(\vec{v}, p + \varepsilon w) - L(\vec{v}, p)] = -c(\vec{v}, w) + (q, w)$$

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., $\vec{u} \in H_0^{\text{div}}$ and $\nabla \cdot \vec{u} = 0$. For the energy functional we have that

$$\begin{aligned} I(\vec{v} + \vec{u}) &= \frac{1}{2} \int_{\Omega} (\vec{v} + \vec{u}) \cdot \mathsf{K}^{-1}(\vec{v} + \vec{u}) d\vec{x} = I(\vec{v}) + I(\vec{u}) + b(\vec{u}, \vec{v}) \\ &= I(\vec{v}) + I(\vec{u}) + c(\vec{u}, p) = I(\vec{v}) + I(\vec{u}) > I(\vec{v}), \end{aligned}$$

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 $L(\vec{v}, w) \leq L(\vec{v}, p) \leq L(\vec{u}, p)$ for all $\vec{u} \neq \vec{v}$ and $w \neq p$. The right inequality can be shown as follows

$$\begin{aligned} L(\vec{v} + \vec{u}, p) &= \frac{1}{2} b(\vec{v} + \vec{u}, \vec{v} + \vec{u}) - c(\vec{v} + \vec{u}, p) + (q, p) \\ &= L(\vec{v}, p) + I(\vec{u}) + b(\vec{u}, \vec{v}) - c(\vec{u}, p) = L(\vec{v}, p) + I(\vec{u}) > L(\vec{v}, p). \end{aligned}$$

The left inequality follows in a similar manner.

6.1.2 Discrete formulation

To discretize the mixed formulation, (6.5), we introduce a grid $\Omega_h = \cup \Omega_i$ and replace $L^2(\Omega)$ and $H_0^{\text{div}}(\Omega)$ by finite-dimensional subspaces U and V defined over the grid. Likewise, we rewrite the inner products (6.6) as sums of integrals localized to cells

$$\begin{aligned} b(\cdot, \cdot)_h : V \times V &\rightarrow \mathbb{R}, & b(\vec{u}, \vec{v})_h &= \sum_i \int_{\Omega_i} \vec{u} \cdot \mathsf{K}^{-1} \vec{v} d\vec{x} \\ c(\cdot, \cdot)_h : V \times U &\rightarrow \mathbb{R}, & c(\vec{u}, p)_h &= \sum_i \int_{\Omega_i} p \nabla \cdot \vec{u} d\vec{x} \\ (\cdot, \cdot)_h : U \times U &\rightarrow \mathbb{R}, & (q, w)_h &= \sum_i \int_{\Omega_i} q w d\vec{x}, \end{aligned} \tag{6.8}$$

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 [221, 46] of lowest order for triangular, tetrahedral or regular parallelepiped grids, $L^2(\Omega)$ is replaced by

$$U = \{p \in L^2(\cup \Omega_i) : p|_{\Omega_i} \text{ is constant } \forall \Omega_i \subset \Omega\} = \text{span}\{\chi_i\},$$

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, $H_0^{\text{div}}(\Omega)$ is replaced by a space V consisting of functions $\vec{v} \in H_0^{\text{div}}(\cup \Omega_i)$ 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

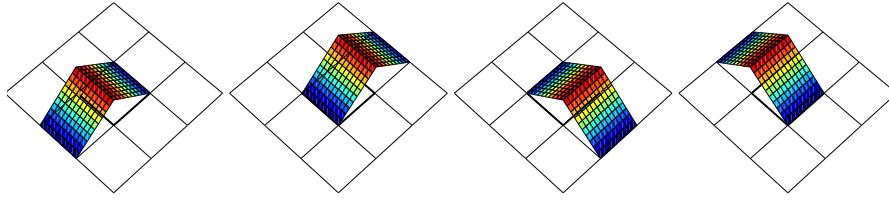


Fig. 6.3. Illustration of the velocity basis for the lowest-order Raviart–Thomas (RT0) method for a 2D Cartesian grid. There are four basis functions that have nonzero support in the interior of the center cell. The basis functions correspond to degrees-of-freedom associated with the west, east, south, 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

$$\vec{\psi}_{ik} \in \mathcal{P}_1(\Omega_i)^d \cup \mathcal{P}_1(\Omega_k)^d \quad \text{and} \quad (\vec{\psi}_{ik} \cdot \vec{n}_{jl})|_{\Gamma_{jl}} = \begin{cases} 1 & \text{if } \Gamma_{jl} = \Gamma_{ik}, \\ 0 & \text{otherwise,} \end{cases}$$

Figure 6.3 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 $\mathbf{p} = \{p_i\}$ and $\mathbf{v} = \{v_{ik}\}$. Using the same functions as test functions, we derive a linear system of the form

$$\begin{bmatrix} \mathbf{B} & -\mathbf{C}^T \\ \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{q} \end{bmatrix}. \quad (6.9)$$

Here, $\mathbf{B} = [b_{ik,jl}]$, $\mathbf{C} = [c_{i,kl}]$, and $\mathbf{q} = [q_i]$, where:

$$\begin{aligned} b_{ik,jl} &= b(\vec{\psi}_{ik}, \vec{\psi}_{jl})_h = \sum_{\ell} \int_{\Omega_{\ell}} \vec{\psi}_{ik} \cdot \mathbf{K}^{-1} \vec{\psi}_{jl} d\vec{x}, \\ c_{i,kl} &= c(\vec{\psi}_{kl}, \chi_i)_h = \int_{\Omega_i} \nabla \cdot \vec{\psi}_{kl} d\vec{x}, \\ q_i &= (q, \chi_i)_h = \int_{\Omega_i} q d\vec{x}. \end{aligned} \quad (6.10)$$

Note that for the first-order Raviart–Thomas finite elements, we have

$$c_{i,kl} = \begin{cases} 1, & \text{if } i = k, \\ -1, & \text{if } i = l, \\ 0, & \text{otherwise.} \end{cases}$$

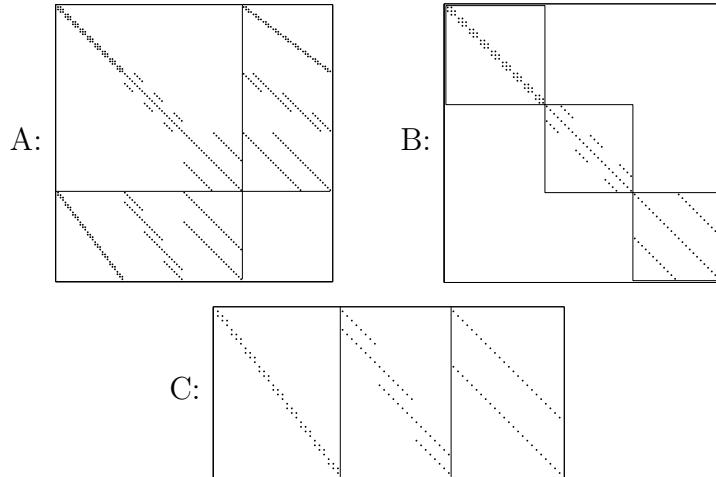


Fig. 6.4. Sparsity patterns for the full mixed finite-element matrix \mathbf{A} , and the matrix blocks \mathbf{B} and \mathbf{C} for a $4 \times 3 \times 3$ Cartesian grid.

The matrix entries $b_{ik,jl}$ depend on the geometry of the grid cells and whether \mathbf{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 \mathbf{K} -orthogonal.

In [4], 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 \mathbf{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 6.4 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 (6.9). The matrix blocks \mathbf{B} and \mathbf{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 [4], 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 6.3.4. 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.

6.1.3 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 (6.9) 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 (6.3) 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 6.1.1). 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 $(\vec{v}, p, \pi) \in H_0^{\text{div}}(\Omega_h) \times L^2(\Omega_h) \times H^{\frac{1}{2}}(\Gamma_h)$ such that

$$\begin{aligned} \sum_i \int_{\Omega_i} (\vec{u} \cdot \mathbf{K}^{-1} \vec{v} - p \nabla \cdot \vec{u}) d\vec{x} + \sum_{ik} \int_{\Gamma_{ik}} \pi \vec{u} \cdot \vec{n} ds &= 0, \\ \sum_i \int_{\Omega_i} w \nabla \cdot \vec{v} d\vec{x} &= \int_{\Omega_h} q w d\vec{x}, \\ \sum_{ik} \int_{\Gamma_{ik}} \mu \vec{v} \cdot \vec{n} ds &= 0 \end{aligned} \quad (6.11)$$

for all test functions $\vec{u} \in H_0^{\text{div}}(\Omega_h)$, $w \in L^2(\Omega_h)$, 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 traces¹ 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 (6.11) in a more compact form,

$$\begin{aligned} b(\vec{u}, \vec{v}) - c(\vec{u}, p) + d(\vec{u}, \pi) &= 0 \\ c(\vec{v}, w) &= (q, w) \\ d(\vec{v}, \mu) &= 0, \end{aligned} \quad (6.12)$$

where the inner products $b(\cdot, \cdot)$, $c(\cdot, \cdot)$, and (\cdot, \cdot) are defined as in (6.6), and $d(\cdot, \cdot)$ is a new inner product defined over the interior faces,

$$d(\cdot, \cdot)_h : H_0^{\text{div}}(\Omega_h) \times H^{\frac{1}{2}}(\Gamma_h) \rightarrow \mathbb{R}, \quad d(\vec{v}, \pi) = \sum_{ik} \int_{\Gamma_{ik}} \pi \vec{v} \cdot \vec{n} \, ds. \quad (6.13)$$

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,

$$\Pi = \text{span}\{\mu_{ik}\}, \quad \mu_{ik}(\vec{x}) = \begin{cases} 1, & \text{if } \vec{x} \in \Gamma_{ik}, \\ 0, & \text{otherwise.} \end{cases} \quad (6.14)$$

Using these basis functions as test and trial functions, one can derive a discrete linear system of the form,

$$\begin{bmatrix} \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \mathbf{C}^T & \mathbf{0} & \mathbf{0} \\ \mathbf{D}^T & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ -\mathbf{p} \\ \boldsymbol{\pi} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{q} \\ \mathbf{0} \end{bmatrix}, \quad (6.15)$$

where the vectors \mathbf{v} , \mathbf{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 \mathbf{B} and \mathbf{C} are defined as in (6.10), and \mathbf{D} has two non-zero entries per column (one entry for each side of the cell interfaces).

The linear system (6.15) is an example of a sparse, symmetric, indefinite system (i.e., a system \mathbf{A} whose quadratic form $\mathbf{x}^T \mathbf{A} \mathbf{x}$ 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 (6.15) to form a positive-definite system (the Schur complement) for the face pressures,

$$(\mathbf{D}^\top \mathbf{B}^{-1} \mathbf{D} - \mathbf{F}^\top \mathbf{L}^{-1} \mathbf{F}) \boldsymbol{\pi} = \mathbf{F}^\top \mathbf{L}^{-1} \mathbf{q}. \quad (6.16)$$

Here, $\mathbf{F} = \mathbf{C}^\top \mathbf{B}^{-1} \mathbf{D}$ and $\mathbf{L} = \mathbf{C}^\top \mathbf{B}^{-1} \mathbf{C}$. Given the face pressures, the cell pressures and fluxes can be reconstructed by back-substitution, i.e., by solving

$$\mathbf{L}\mathbf{p} = \mathbf{q} + \mathbf{F}\boldsymbol{\pi}, \quad \mathbf{B}\mathbf{u} = \mathbf{C}\mathbf{p} - \mathbf{D}\boldsymbol{\pi}. \quad (6.17)$$

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 [44, 41, 46]. 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.

6.2 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

$$\mathbf{v}_i = \mathbf{T}_i(\mathbf{e}\mathbf{p}_i - \boldsymbol{\pi}_i), \quad (6.18)$$

where \mathbf{v}_i is a vector of all fluxes associated with a cell Ω_i , $\mathbf{e}_i = (1, \dots, 1)^\top$ has one unit value per face of the cell, $\boldsymbol{\pi}_i$ is a vector of face pressures, and \mathbf{T}_i is a matrix of one-sided transmissibilities. The local discretization (6.18) can alternatively be written as

$$\mathbf{M}_i \mathbf{v}_i = \mathbf{e}\mathbf{p}_i - \boldsymbol{\pi}_i. \quad (6.19)$$

Consistent with the discussion in Section 6.1, the matrix \mathbf{M} is referred to as the local inner product. From the local discretizations (6.18) or (6.19) 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., [81].

Going back to the TPFA method formulated in Section 4.4.1, it follows immediately from (4.51) that the method can be written as in (6.18) and that the resulting \mathbf{T}_i matrix is diagonal with entries

$$(\mathbf{T}_i)_{kk} = \vec{n}_{ik} \cdot \mathbf{K} \vec{c}_{ik} / |\vec{c}_{ik}|^2, \quad (6.20)$$

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 (6.19) follows trivially by setting $\mathbf{M}_i = \mathbf{T}_i^{-1}$. Examples of consistent methods that can be written in the form (6.18) or (6.19) include the lowest-order Raviart–Thomas methods seen in the previous section, multipoint flux approximation (MPFA) schemes [8, 80, 6], and recently developed mimetic finite-difference methods [47]. For all these methods, the corresponding \mathbf{T}_i and \mathbf{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 (6.19) with flux and pressure continuity across cell faces. By assembling the contributions from all cells in the grid, we get the following linear system

$$\begin{bmatrix} \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \mathbf{C}^\top & \mathbf{0} & \mathbf{0} \\ \mathbf{D}^\top & \mathbf{0} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{v} \\ -\mathbf{p} \\ \boldsymbol{\pi} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{q} \\ \mathbf{0} \end{bmatrix}, \quad (6.21)$$

where the first row in the block-matrix equation corresponds to (6.19) for all grid cells. The vector \mathbf{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 \mathbf{p} contains the cell pressures, and $\boldsymbol{\pi}$ the face pressures. The matrices \mathbf{B} and \mathbf{C} are block diagonal with each block corresponding to a cell,

$$\mathbf{B} = \begin{bmatrix} \mathbf{M}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{M}_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \dots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{M}_n \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} \mathbf{e}_1 & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{e}_2 & \dots & \mathbf{0} \\ \vdots & \vdots & \dots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{e}_n \end{bmatrix}, \quad (6.22)$$

where $\mathbf{M}_i = \mathbf{T}_i^{-1}$. Similarly, each column of \mathbf{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, \mathbf{D} has a single nonzero entry in each column that corresponds to an interface between a cell and the exterior.

The hybrid system (6.21) 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 6.1.3, and then solved using either MATLAB’s standard linear solvers or a highly-efficient, third-party solver such as the agglomeration multigrid solver, AGMG [203, 10]. From the expressions in (6.16) and (6.17), 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 \mathbf{B}^{-1} in the solution procedure above. Moreover, the matrix \mathbf{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 \mathbf{B}^{-1} matrix. Thus, (6.21) encompasses a family of discretization schemes whose properties are determined by the choice of \mathbf{B} , which we will discuss in more detail in Section 6.3.

However, before digging into details about specific, consistent methods, we revisit the example discussed on page 181 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 6.2. 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 \mathbf{T}_i or its equivalent inner product $\mathbf{M}_i = \mathbf{T}_i^{-1}$ 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 = incompMimetic(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

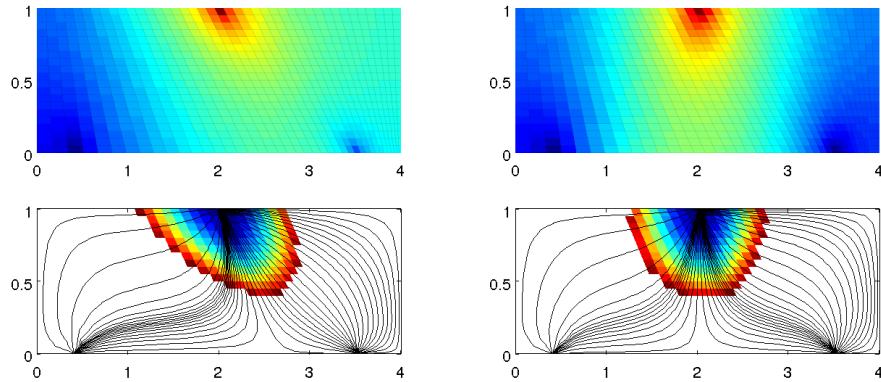


Fig. 6.5. Solution of a symmetric flow problem in a homogeneous domain using the TPFA method (left) and the mimetic method (right) on a skew grid that is not K -orthogonal.

a TPFA-type system.) In Figure 6.5 we have applied the mimetic solver to the exact same setup as in Figure 6.2. 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 6.2 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.

COMPUTER EXERCISES:

36. Run the `simpleBCmimetic` example from the `mimetic` module to familiarize yourself with the mimetic solver, the mixed-hybrid formulation, the Schur-component reduction, etc.
37. Can you cure the grid-orientation problems observed in Figure 5.7 on page 172 by using a consistent discretization?
38. Compare the solutions computed by the TPFA and the mimetic method for the SAIGUP model as set up in Section 5.4.4 on page 176. In particular, you should compare discrepancies in pressure and time-of-flight values at the well perforations. Which solution do you trust most?
Hints: (i) Remember to use the correct inner product when setting up well models. (ii) The mimetic linear system is much larger than the TPFA system and you may have to use an iterative solver like AGMG to reduce runtime and memory consumption.

6.3 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., [36].

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 (6.18) or (6.19) and are constructed so that they are exact for linear pressure fields and give a symmetric positive-definite matrix \mathbf{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 $\vec{v} = -\mathbf{K}\vec{a}$. 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 4.10. Using Darcy's law and the notion of a one-sided transmissibility \mathbf{T}_{ik} (which will generally not be the same as the two-point transmissibility), the flux and pressure drop can be related as follows,

$$v_{ik} = -\vec{n}_{ik} \mathbf{K} \vec{a} = T_{ik}(p_i - \pi_{ik}) = -T_{ik} \vec{c}_{ik} \cdot \vec{a}. \quad (6.23)$$

To get this relations in the one of the local forms (6.18) and (6.19) we collect all vectors \vec{c}_{ik} and \vec{n}_{ik} defined for a specific cell Ω_i as rows in two matrices \mathbf{C}_i and \mathbf{N}_i . Because the relation (6.23) is required to hold for all linear pressure drops, i.e., for an arbitrary vector \vec{a} , we see that the matrices \mathbf{M}_i and \mathbf{T}_i must satisfy the following consistency conditions

$$\mathbf{M} \mathbf{N} \mathbf{K} = \mathbf{C}, \quad \mathbf{N} \mathbf{K} = \mathbf{T} \mathbf{C}, \quad (6.24)$$

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 \mathbf{M}_i , or equivalently the local transmissibility matrix \mathbf{T}_i , is positive

definite and satisfies (6.24) will be a consistent, first-order accurate discretization. In the rest of the section, we will discuss various choices of valid inner products \mathbf{M}_i or reverse inner products \mathbf{T}_i .

6.3.1 General family of inner products

The original article [47] 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 \mathbf{A} is a diagonal matrix with element A_{jj} equal the are of the j -th face, then the flux inner product \mathbf{M}_{flux} is related to the velocity inner product \mathbf{M}_{vel} through

$$\mathbf{M}_{\text{flux}} = \mathbf{A}^{-1} \mathbf{M}_{\text{vel}} \mathbf{A}^{-1}. \quad (6.25)$$

To simplify the derivation of valid solutions, we start by stating the following geometrical property which relates \mathbf{C} and \mathbf{N} as follows (for a proof, see [47]):

$$\mathbf{C}^T \mathbf{N} = \mathbf{V} = \text{diag}(|\Omega_i|). \quad (6.26)$$

Next, we multiply the left matrix equation in (6.24) by $\mathbf{K}^{-1} \mathbf{C}^T \mathbf{N}$

$$\mathbf{C}(\mathbf{K}^{-1} \mathbf{C}^T \mathbf{N}) = (\mathbf{M} \mathbf{N} \mathbf{K})(\mathbf{K}^{-1} \mathbf{C}^T \mathbf{N}) = \mathbf{M} \mathbf{N} \mathbf{C}^T \mathbf{N} = \mathbf{M} \mathbf{N} \mathbf{V},$$

from which it follows that there is a family of valid solution that has the form

$$\mathbf{M} = \frac{1}{|\Omega_i|} \mathbf{C} \mathbf{K}^{-1} \mathbf{C}^T + \mathbf{M}_2, \quad (6.27)$$

where \mathbf{M}_2 is a matrix defined such that $\mathbf{M}_2 \mathbf{N} = \mathbf{0}$, i.e., any matrix whose rows lie in the left nullspace of \mathbf{N}^T . Moreover, to make a sensible method we must require that \mathbf{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,

$$\begin{aligned} \mathbf{M} &= \frac{1}{|\Omega_i|} \mathbf{C} \mathbf{K}^{-1} \mathbf{C}^T + \mathbf{Q}_N^\perp \mathbf{S}_M \mathbf{Q}_N^\perp{}^T \\ &= \frac{1}{|\Omega_i|} \mathbf{C} \mathbf{K}^{-1} \mathbf{C}^T + \mathbf{P}_N^\perp \mathbf{S}_M \mathbf{P}_N^\perp, \end{aligned} \quad (6.28)$$

where \mathbf{S}_M denotes any symmetric, positive definite matrix, \mathbf{Q}_N^\perp is an orthonormal basis for the left null space of \mathbf{N}^T , and \mathbf{P}_N^\perp is the null-space projection $\mathbf{I} - \mathbf{Q}_N \mathbf{Q}_N{}^T$ in which \mathbf{Q}_N is a basis for the spaces spanned by the columns of

\mathbf{N} . Similarly, we can derive a closed expression for the inverse inner product \mathbf{T} by multiplying the right matrix equation in (6.24) by \mathbf{V}^{-1} from the left and by \mathbf{V} from the right and using the identity (6.26),

$$\mathbf{T}\mathbf{C} = \mathbf{V}^{-1}(\mathbf{N}\mathbf{K})\mathbf{V} = \mathbf{V}^{-1}(\mathbf{N}\mathbf{K})(\mathbf{C}^\top\mathbf{N})^\top = \mathbf{V}^{-1}(\mathbf{N}\mathbf{K}\mathbf{N}^\top)\mathbf{C}$$

By the same argument as for \mathbf{M} , mutatis mutandis, we obtain the following family of inverse inner products,

$$\begin{aligned}\mathbf{T} &= \frac{1}{|\Omega_i|}\mathbf{N}\mathbf{K}\mathbf{N}^\top + \mathbf{Q}_C^\perp\mathbf{S}_T\mathbf{Q}_C^{\perp\top} \\ &= \frac{1}{|\Omega_i|}\mathbf{N}\mathbf{K}\mathbf{N}^\top + \mathbf{P}_C^\perp\mathbf{S}_T\mathbf{P}_C^\perp,\end{aligned}\tag{6.29}$$

where \mathbf{Q}_C^\perp is an orthonormal basis for the left nullspace of \mathbf{C}^\top and $\mathbf{P}_C^\perp = \mathbf{I} - \mathbf{Q}_C\mathbf{Q}_C^\top$ is the corresponding nullspace projection.

The matrices \mathbf{M} and \mathbf{T} in (6.28) and (6.29) are evidently symmetric, so we only need to prove that they are positive definite. We start by writing the inner product as $\mathbf{M} = \mathbf{M}_1 + \mathbf{M}_2$, and observe that each of these matrices are positive semi-definite and our result follows if we can prove that this implies that \mathbf{M} is positive definite. Let \mathbf{z} be an arbitrary nonzero vector which we split uniquely as $\mathbf{z} = \mathbf{N}\mathbf{x} + \mathbf{y}$, where \mathbf{x} lies in the column space of \mathbf{N} and \mathbf{y} lies in the left nullspace of \mathbf{N}^\top . If \mathbf{y} is zero, we have

$$\begin{aligned}\mathbf{z}^\top\mathbf{M}\mathbf{z} &= \mathbf{x}^\top\mathbf{N}^\top\mathbf{M}_1\mathbf{N}\mathbf{x} \\ &= |\Omega_i|^{-1}\mathbf{x}^\top(\mathbf{N}^\top\mathbf{C})\mathbf{K}^{-1}(\mathbf{C}^\top\mathbf{N})\mathbf{x} = |\Omega_i|\mathbf{x}^\top\mathbf{K}^{-1}\mathbf{x} > 0\end{aligned}$$

because \mathbf{K}^{-1} is a positive definite matrix. If \mathbf{y} is nonzero, we have

$$\mathbf{z}^\top\mathbf{M}\mathbf{z} = \mathbf{z}^\top\mathbf{M}_1\mathbf{z} + \mathbf{y}^\top\mathbf{M}_2\mathbf{y} > 0$$

because $\mathbf{z}^\top\mathbf{M}_1\mathbf{z} \geq 0$ and $\mathbf{y}^\top\mathbf{M}_2\mathbf{y} > 0$. An analogous argument holds for the matrix \mathbf{T} , and hence we have proved that (6.28) and (6.29) give a well-defined family of consistent discretizations.

So far, we have not put any restrictions on the matrices \mathbf{S}_M and \mathbf{S}_T that will determine the specific methods, except for requiring that they should be positive definite. In addition, these matrices should mimic the scaling properties of the continuous equation, which is invariant under affine transformations of space and permeability,

$$\vec{x} \mapsto \boldsymbol{\sigma}\vec{x} \quad \text{and} \quad \mathbf{K} \mapsto \boldsymbol{\sigma}^\top\mathbf{K}\boldsymbol{\sigma},\tag{6.30}$$

To motivate how we should choose the matrices \mathbf{S}_M and \mathbf{S}_T to mimic these scaling properties, we will look at a simple 1D example:

Example 6.1. Consider the a grid cell $x \in [-1, 1]$, for which $\mathbf{N} = \mathbf{C} = [1, -1]^\top$ and $\mathbf{Q}_N^\perp = \mathbf{Q}_C^\perp = \frac{1}{\sqrt{2}}[1, 1]^\top$. Hence

$$\begin{aligned}\mathbf{M} &= \frac{1}{2} \begin{bmatrix} 1 \\ -1 \end{bmatrix} \frac{1}{K} [1, -1] + \frac{1}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} S_M [1, 1], \\ \mathbf{T} &= \frac{1}{2} \begin{bmatrix} 1 \\ -1 \end{bmatrix} K [1, -1] + \frac{1}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} S [1, 1].\end{aligned}\quad (6.31)$$

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

$$\begin{aligned}\mathbf{M} &= \frac{1}{2K} \left(\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} + \frac{2}{t} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \right), \\ \mathbf{T} &= \frac{K}{2} \left(\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} + \frac{t}{2} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \right).\end{aligned}\quad (6.32)$$

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 [153].

6.3.2 General parametric family

Motivated by the example above, we propose to choose the matrix \mathbf{S}_T as the diagonal of the first matrix term in (6.29) so that these two terms scale similarly under transformations of the type (6.30). Using this definition of \mathbf{S}_T and that \mathbf{M} should be equal to \mathbf{T}^{-1} suggests the following general family of inner products that only differ in the constant in front of the second (regularization) matrix term:

$$\begin{aligned}\mathbf{M} &= \frac{1}{|\Omega_i|} \mathbf{C} \mathbf{K}^{-1} \mathbf{C}^\top + \frac{|\Omega_i|}{t} \mathbf{P}_N^\perp \text{diag}(\mathbf{N} \mathbf{K} \mathbf{N}^\top)^{-1} \mathbf{P}_N^\perp, \\ \mathbf{T} &= \frac{1}{|\Omega_i|} \left[\mathbf{N} \mathbf{K} \mathbf{N}^\top + t \mathbf{P}_C^\perp \text{diag}(\mathbf{N} \mathbf{K} \mathbf{N}^\top) \mathbf{P}_C^\perp \right].\end{aligned}\quad (6.33)$$

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.

6.3.3 Two-point type methods

A requirement of the two-point method is that the transmissibility matrix \mathbf{T} (and hence also the matrix \mathbf{M} of the inner product) should be diagonal.

Looking at (6.24), we see that this is only possible if the vectors $K\vec{N}_{ik}$ 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 6. An explicit expression for the diagonal entries of the two-point method on K -orthogonal grids has already been given in (6.20). 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 6.2. We consider the grid cell $[-1, 1] \times [-1, 1]$ in 2D and calculate the components that make up the inverse inner product \mathbf{T} for a diagonal and a full permeability tensor \mathbf{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 (6.20), 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 $\mathbf{T}=\text{diag}([2 2 2 2])$ 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 \mathbf{S}_T that scales similarly to the first term in the definition of \mathbf{T} in (6.29), we could choose \mathbf{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:

$$\begin{array}{cccc} W = & 1 & -1 & 0 \\ & -1 & 1 & 0 \\ & 0 & 0 & 1 \\ & 0 & 0 & -1 \end{array} \quad \begin{array}{cccc} P*St*P = & 0.5 & 0.5 & 0 \\ & 0.5 & 0.5 & 0 \\ & 0 & 0 & 0.5 \\ & 0 & 0 & 0.5 \end{array}$$

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,

$$\begin{array}{cccc} T1 = & 2.0 & 0 & 0 \\ & 0 & 2.0 & 0 \\ & 0 & 0 & 2.0 \\ & 0 & 0 & 0 \end{array} \quad \begin{array}{cccc} T2 = & 2.0 & 0 & 0.5 \\ & 0 & 2.0 & -0.5 \\ & 0.5 & -0.5 & 2.0 \\ & -0.5 & 0.5 & 0 \end{array}$$

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:

$$\begin{aligned} M &= \frac{1}{|\Omega_i|} CK^{-1} C^T + \frac{|\Omega_i|}{2} P_N^\perp \text{diag}(NKN^T)^{-1} P_N^\perp, \\ T &= \frac{1}{|\Omega_i|} [NKN^T + 2 P_C^\perp \text{diag}(NKN^T) P_C^\perp]. \end{aligned} \quad (6.34)$$

The observant reader will notice that this is a special case for $t = 2$ of the general family (6.33) 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

$$S = \text{computeMimeticIP}(G, \text{rock}, \text{'InnerProduct'}, \text{'ip_quasitpf'});$$

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:

$$S = \text{computeMimeticIP}(G, \text{rock}, \text{'InnerProduct'}, \text{'ip_tpf'});$$

6.3.4 Raviart–Thomas type inner product

To compute the mixed finite-element inner product on cells that are not simplices 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 6.3. 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, $\mathbf{K} = \mathbf{I}$. This entails careful treatment of a sticky point: whereas the mixed inner product involves velocities, the mimetic inner product involves fluxes that 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:

$$\begin{aligned} \int_0^1 \int_0^1 \vec{\psi}_1 \cdot \vec{\psi}_1 dx dy &= \int_0^1 (x-1)^2 dx = \frac{1}{3} \\ \int_0^1 \int_0^1 \vec{\psi}_1 \cdot \vec{\psi}_2 dx dy &= \int_0^1 (x-1)x dx = -\frac{1}{6} \end{aligned}$$

This means that the RT0 inner product in 2D reads

$$\mathbf{M} = \begin{bmatrix} \frac{1}{3} & -\frac{1}{6} & 0 & 0 \\ -\frac{1}{6} & \frac{1}{3} & 0 & 0 \\ 0 & 0 & \frac{1}{3} & -\frac{1}{6} \\ 0 & 0 & -\frac{1}{6} & \frac{1}{3} \end{bmatrix}$$

Similarly as in Example 6.2, 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\Q'*Q)/vol
M2 = P*Sm*P

```

The result of this computation is

$$\begin{array}{ccccccccc}
M1 & = & 0.25 & -0.25 & 0 & 0 & M2 & = & 0.50 \quad 0.50 \quad 0 \quad 0 \\
& & -0.25 & 0.25 & 0 & 0 & & 0.50 & 0.50 \quad 0 \quad 0 \\
& & 0 & 0 & 0.25 & -0.25 & & 0 & 0 \quad 0.50 \quad 0.50 \\
& & 0 & 0 & -0.25 & 0.25 & & 0 & 0 \quad 0.50 \quad 0.50
\end{array}$$

from which it follows that 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 (6.30). The quasi-RT0 inner product reads

$$\begin{aligned}
M &= \frac{1}{|\Omega_i|} CK^{-1} C^T + \frac{|\Omega_i|}{6} P_N^\perp \text{diag}(NKN^T)^{-1} P_N^\perp, \\
T &= \frac{1}{|\Omega_i|} [NKN^T + 6P_C^\perp \text{diag}(NKN^T)P_C^\perp].
\end{aligned} \tag{6.35}$$

and is a special case of the general family (6.33) 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.

6.3.5 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,

$$T = \frac{1}{|\Omega_i|} [NKN^T + \frac{6}{d} \text{tr}(K) A (I - Q Q^T) A]. \tag{6.36}$$

where A is the diagonal matrix containing face areas and Q is an orthogonal basis for the range of AC . The inner product, referred to as 'ip_simple', was

inspired by [47] and introduced in [5] 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 \mathbf{C} .

6.3.6 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 [8, 80, 6]. 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 [136, 165] 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 formulation of the MPFA-O method can be used with a calling sequence that is similar to that of the TPFA and the mimetic methods:

```
mrstModule add mpfa;
hT = computeMultiPointTrans(G, rock);
state = incompMPFA(state, G, hT, fluid)
```

By default, `computeMultiPointTrans` uses MATLAB to invert a sequence of small matrices that arise in the discretization. This operation is relatively slow and the routine therefore takes an optional parameter '`invertBlocks`' that can be set equal to '`MEX`' to invoke C-accelerated for the inversion of the small matrix blocks. This option may not work if MRST has not been properly set up to automatically compile C-code on your computer.

6.3.7 Monotonicity

The main motivation for developing consistent discretizations is to get schemes that are convergent for rough grids and full-tensor permeabilities and less influenced by grid-orientation errors. All consistent methods discussed above fulfill this requirement. On the other hand, these methods are all linear and not guaranteed to satisfy discrete counterparts of the maximum principle fulfilled by the continuous solution. In other words, the discrete solutions may contain oscillations that are not present in the underlying physical system, particularly for cases with full-tensor permeabilities and large anisotropy ratios and/or high aspect ratio grids. While such oscillations are not very serious for single-phase, incompressible flow, they may have a significant impact on multiphase flows that can be highly sensitive to small pressure variations. As an example, think of an oil-gas system operating near the bubble point. If the numerical approximation is below the bubble point while the true pressure is above, the simulation will incorrectly predict liberation of free gas, or vice versa, which may lead to highly unstable and divergent behavior. Two-point discretizations, on the other hand, are strictly monotone if all transmissibilities are non-negative.

For brevity, we will only present an example that illustrates lack of monotonicity for linear, consistent schemes. We refer the interested reader to [200] and references therein for a thorough discussion of the underlying mathematical properties that cause these unphysical oscillations.

Example 6.4 (Example 2 from [153]). To illustrate the different behavior of the TPFA and linear, consistent discretizations, we consider a 2D domain of size $n \times n$, where n is also the number of grid cells in each spatial direction. To generate a non-orthogonal grid, we use the `twister` routine, which you have encountered multiple times already. This routine normalises all coordinates in a rectilinear grid to the interval $[0, 1]$, then perturbs all interior points by adding $0.03 \sin(\pi x) \sin(3\pi(y - 1/2))$ to the x -coordinates and subtracting the same value from the y -coordinate before transforming back to the original domain. This creates a non-orthogonal, but smoothly varying, logically Cartesian grid. The domain has Dirichlet boundary conditions on the left and right-hand sides, with values one and zero, respectively, and zero Neumann conditions at the top and bottom.

An important design principle for consistent discretizations is that they, unlike the TPFA method, should reproduce flows that result from a linear pressure drop exactly on grids that are not K-orthogonal. We therefore start by considering a diagonal permeability tensor with anisotropy ratio 1 : 1000 aligned with the x -axis. Figure 6.6 compares the discrete solutions computed by the TPFA method, the local-flux mimetic (MPFA-O) method, and the mimetic method with inner product ‘ip_simple’ on a 101×101 grid. Whereas the mimetic and MPFA-O schemes produce the expected result, the pressure and velocity solutions computed by TPFA show significant grid-orientation effects and are obviously wrong.

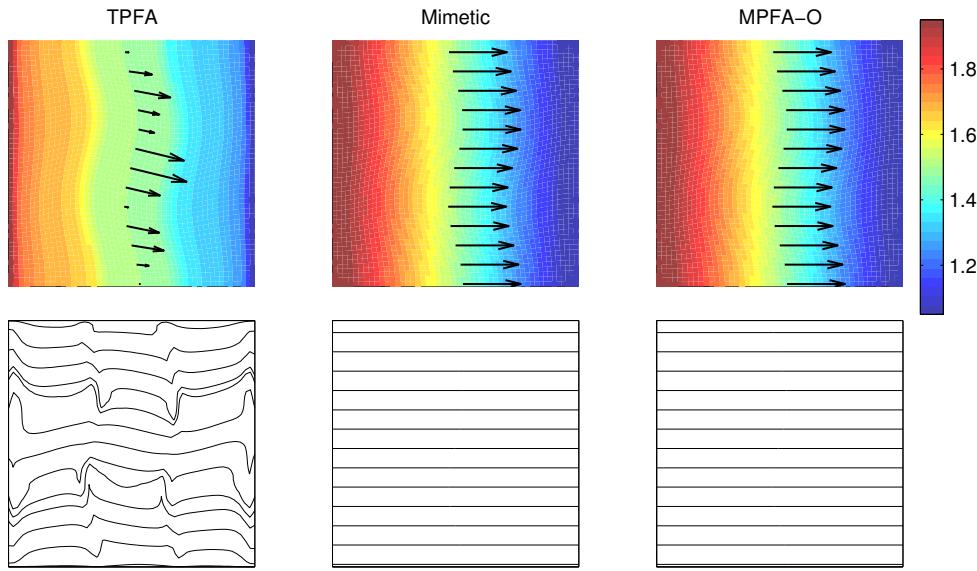


Fig. 6.6. Grid-orientation effects for a flow described by a linear pressure drop from the left to the right boundary. Approximate solutions computed by three different incompressible solvers from the `incomp`, `mimetic`, and `mpfa` modules, respectively. The permeability field is homogeneous with anisotropy ratio 1 : 1000 aligned with the grid.

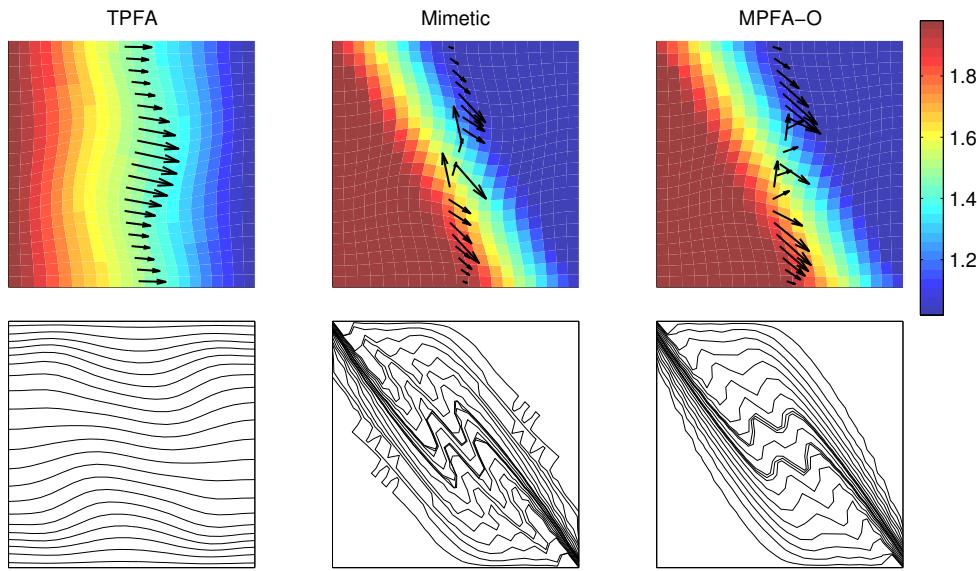


Fig. 6.7. Monotonicity illustrated with the same setup as in Figure 6.7, but with the anisotropy ratio 1 : 1000 making an angle $\pi/6$ with the x -direction.

To illustrate non-monotonic behavior, we repeat the experiment with the tensor rotated by $\pi/6$ on a grid with 21×21 cells, see Figure 6.7. The solution computed by the TPFA method is monotone, but fails to capture the correct flow pattern in which streamlines starting near the upper-left corner should exit near the lower-right corner. Judging from the color plots, the pressure values computed by the mimetic and the MPFA-O methods appear to be correct and have no discernible oscillations. However, both the streamline distribution and the velocity vectors plotted for cells along the midsection of the grid clearly show that the resulting velocity fields are highly oscillatory, in particular for the mimetic method. If we change the inner product from its default value ('ip_simple') to e.g., the quasi two-point inner product ('ip_quasitpf'), the solution is qualitatively the same as for the MFPA-O method.

COMPUTER EXERCISES:

39. The **mimetic** module contains two examples, **simpleTPFA** and **secondTPFA**, that compare the TPFA method implemented in the **incomp** module with its mimetic counterpart. Do the methods produce the same results if you perturb the grid cells using **twister** or specify an anisotropic permeability field? What happens if you use a different inner product?
40. A standard method to construct analytical solutions to the Laplacian equation in 2D is to write the unknown as the real or imaginary part of an complex analytic function $f(z)$. We choose the function $f(z) = (z + \frac{1}{2}z^2)$ and set our analytical solution to be $p_a(x, y) = \text{Im}(f(x + iy)) = y + xy$. By prescribing $p_a(x, y)$ along the boundary, we obtain a Dirichlet problem with known solution.
Use this technique to study the convergence of the TPFA method and at least two different consistent methods on a sequence of rough non-Cartesian grids of increasing resolution. Try to compare grids with cells that are always non-Cartesian at all resolutions with grids in which the cells are non-Cartesian at coarse resolutions but gradually approach a Cartesian shape as the grid is refined.

Single-Phase Flow and Rapid Prototyping

In previous chapters we have outlined and explained in detail how to discretize and solve incompressible flow problems. This chapter will teach you how to discretize the basic equations for single-phase, compressible flow. To this end, we will rely heavily on the library for automatic differentiation (AD), which was briefly discussed in Section A.5, and the discrete differential and averaging operators that were introduced in Section 4.4.2. As briefly shown in Examples 4.2 and 4.3 in the same section, these discrete operators enable you to implement discretized flow equations in a form that is compact and close to their mathematical description. Use of automatic differentiation then ensures that no analytical derivatives have to be programmed explicitly as long as the discrete flow equations and constitutive relationships are implemented as a sequence of algebraic operations. In MRST, discrete operators and automatic differentiation are combined with a flexible grid structure, a highly vectorized and interactive scripting language, and a powerful graphical environment. This is in our opinion the main reason why the software has proved to be an efficient tool for developing new computational methods and workflow tools. In this chapter, we try to substantiate this claim by showing several examples of rapid prototyping by first developing a compact and transparent solver for compressible flow, and then extending the basic single-phase model to include pressure-dependent viscosity, non-Newton fluid behavior, and temperature effects. Complete scripts for all the examples can be found in the `ad-1ph` subdirectory of the `book` module.

7.1 Implicit discretization

As our basic model, we consider the single-phase continuity equation,

$$\frac{\partial}{\partial t}(\phi\rho) + \nabla \cdot (\rho\vec{v}) = q, \quad \vec{v} = -\frac{K}{\mu}(\nabla p - g\rho\nabla z). \quad (7.1)$$

The primary unknown is usually the fluid pressure p . 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 4.2. (Notice that q is defined slightly differently in (7.1) than in (4.5)).

Using the discrete operators introduced in Section 4.4.2, the basic implicit discretization of (7.1) reads

$$\frac{(\phi\rho)^{n+1} - (\phi\rho)^n}{\Delta t^n} + \operatorname{div}(\rho v)^{n+1} = q^{n+1}, \quad (7.2a)$$

$$v^{n+1} = -\frac{\mathbf{K}}{\mu^{n+1}} [\operatorname{grad}(p^{n+1}) - g\rho^{n+1} \operatorname{grad}(z)]. \quad (7.2b)$$

Here, $\phi \in \mathbb{R}^{n_c}$ denotes the vector with one porosity value per cell, v is the vector of fluxes per face, and so on. The superscript refers to discrete times at which one wishes to compute the unknown reservoir states and Δt denotes the distance between two such 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 a linear equation in the unknown pressure p^{n+1} , which we can write on residual form as

$$\frac{p^{n+1} - p^n}{\Delta t^n} - \frac{1}{c_t \mu \phi} \operatorname{div}(\mathbf{K} \operatorname{grad}(p^{n+1})) - q^n = 0. \quad (7.3)$$

The assumption of weak compressibility is not always applicable and for generality we assume that ϕ and ρ depend nonlinearly on p so that (7.2) gives rise to a nonlinear system of equations that needs to be solved in each time step. As we will see later in this chapter, the viscosity may also depend on pressure, flow velocity, and/or temperature, which adds further nonlinearity to the system. If we now collect all the discrete equations, we can write the resulting system of nonlinear equations in short vector form as

$$\mathbf{F}(\mathbf{x}^{n+1}; \mathbf{x}^n) = \mathbf{0}, \quad (7.4)$$

where \mathbf{x}^{n+1} is the vector of unknown state variables at the next time step and the vector of current states \mathbf{x}^n can be seen as a parameter.

To solve the nonlinear system (7.4) we will use the Newton–Raphson method (see Example A.4 on page 614): Assume that we have a guess \mathbf{x}_0 and want to move this towards the correct solution, $\mathbf{F}(\mathbf{x}) = \mathbf{0}$. To this end, we write $\mathbf{x} = \mathbf{x}_0 + \Delta\mathbf{x}$ and solve for $\Delta\mathbf{x}$ from the following equation

$$\mathbf{0} = \mathbf{F}(\mathbf{x}_0 + \Delta\mathbf{x}) \approx \mathbf{F}(\mathbf{x}_0) + \nabla\mathbf{F}(\mathbf{x}_0)\delta\mathbf{x}.$$

This gives rise to an iterative scheme in which the approximate solution \mathbf{x}^{i+1} in the $(i+1)$ -th iteration is obtained from

$$\frac{d\mathbf{F}}{d\mathbf{x}}(\mathbf{x}^i)\delta\mathbf{x}^{i+1} = -\mathbf{F}(\mathbf{x}^i), \quad \mathbf{x}^{i+1} \leftarrow \mathbf{x}^i + \delta\mathbf{x}^{i+1}. \quad (7.5)$$

Here, $\mathbf{J} = d\mathbf{F}/d\mathbf{x}$ is the Jacobian matrix, while $\delta\mathbf{x}^{i+1}$ is referred to as the *Newton update* at iteration number $i + 1$. Theoretically, the Newton process exhibits quadratic convergence under certain smoothness and differentiability requirements on \mathbf{F} . Obtaining such convergence in practice, however, will crucially depend on having a sufficiently accurate Jacobian matrix. For complex flow models the computation of residual equations will typically involve evaluation of many constitutive laws that altogether make up complex nonlinear dependencies. Analytical derivation and subsequent coding of the Jacobian can therefore be very time-consuming and prone to human errors. Fortunately, the computation of the Jacobian matrix can in almost all cases be broken down to nested differentiation of elementary operations and functions and is therefore a good candidate for automation using automatic differentiation. This will add an extra computational overhead to your code, but in most cases the added cost is completely offset by the reduced development time. Likewise, unless your model problem is very small, the dominant computational cost of solving a nonlinear PDE comes from the linear solver called within each Newton iteration.

The idea of using automatic differentiation to develop reservoir simulators is not new. This technique was introduced in an early version of the commercial Intersect simulator [70], but has mainly been pioneered through a reimplementation of the GPRS research simulator [50]. The new simulator, called AD-GPRS is primarily based on fully implicit formulations [252, 267, 251] in which independent variables and residual equations are AD structures implemented using ADETL, a library for forward-mode AD realized by expression templates in C++ [265, 264]. This way, the Jacobi matrices needed in the nonlinear Newton-type iterations can be constructed from the derivatives that are implicitly computed when evaluating the residual equations. In [152], the authors discuss how to use the alternative backward-mode differentiation to improve computational efficiency.

7.2 A simulator based on automatic differentiation

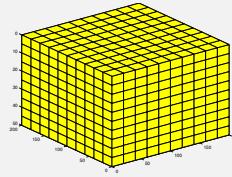
We will now present step-by-step how you can use the AD class in MRST to implement an implicit solver for the compressible, single-phase continuity equation (7.1). In particular, we revisit the discrete spatial differentiation operators from Section 4.4.2 and introduce additional discrete averaging operators that together 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). Starting from this residual form, it is relatively simple to obtain a linearization using automatic differentiation and set up a Newton iteration.

7.2.1 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 = makeRock(G, 30*milli*darcy, 0.3);
```



Beyond this point, our implementation is agnostic to details about the grid, except when we specify well positions on page 209, which would typically have involved a few more code lines for a complex corner-point model like the SAIGUP or the Norne models discussed in Sections 3.5 and 3.3.1.

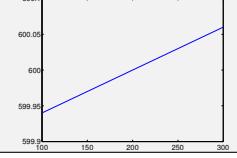
We assume a constant rock compressibility c_r . Accordingly, the pore volume p_v of a grid cell obeys the differential equation¹ $c_r p_v = dp_v/dp$ or

$$p_v(p) = p_{v_r} e^{c_r(p - p_r)}, \quad (7.6)$$

where p_{v_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) );
```

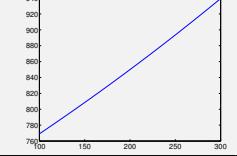


The fluid is assumed to have a constant viscosity, $\mu = 5$ cP. As for the rock, we assume a constant fluid compressibility c resulting in the differential equation $c\rho = d\rho/dp$ for the fluid density. Accordingly,

$$\rho(p) = \rho_r e^{c(p - p_r)}, \quad (7.7)$$

where ρ_r 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

¹ To make a closer correspondence between the computer code and the mathematical equation, we deliberately violate the advice of never using a compound symbol to denote a single mathematical quantity.

```

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-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

```

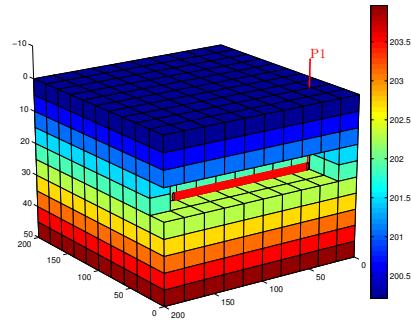


Fig. 7.1. Model with initial pressure and single horizontal well.

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 our simple setup, this differential equation can be solved analytically, but for demonstration purposes, we 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 as shown in Figure 7.1.

7.2.2 Discrete operators and equations

We are now ready to discretize the model. As seen in Section 4.4.2, the discrete version of the gradient operator maps from the set of cells to the set of faces, and for a pressure-field, it computes the pressure difference between neighboring cells of each face. Likewise, the discrete 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 complete code needed to form the

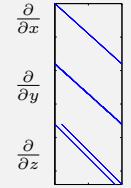
grad and **div** operators has already been presented in Examples 4.2 and 4.3, but here we repeat it to make the example more self-contained.

To define the discrete operators, we need to first compute the map between interior faces and cells

```
C = double(G.faces.neighbors);
intInx = all(C ~= 0, 2);
C = C(intInx, :);
```

Exterior faces need not be included since they will have zero flow because of our assumption of no-flow boundary conditions. It now follows that $\text{grad}(\mathbf{x}) = \mathbf{x}(C(:, 2)) - \mathbf{x}(C(:, 1)) = \mathbf{D}\mathbf{x}$, where \mathbf{D} is a sparse matrix with values ± 1 in columns $C(i, 2)$ and $C(i, 1)$ for row i . 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, (4.57). In addition, we define an *averaging* operator that for each face computes the arithmetic average of the neighboring cells, which we will need to evaluate density values at grid faces:

```
n = size(C,1);
D = sparse([(1:n)'; (1:n)'], C, ...
    ones(n,1)*[-1 1], n, G.cells.num);
grad = @(x) D*x;
div = @(x) -D'*x;
avg = @(x) 0.5 * (x(C(:,1)) + x(C(:,2)));
```



This is all we need to define the spatial discretization for a homogeneous medium on a grid with cubic cells. To make a generic spatial discretization that also can account for more general cell geometries and heterogeneities, we need to include transmissibilities. To this end, we first compute one-sided transmissibilities $T_{i,j}$ using the function **computeTrans**, which was discussed in detail in Section 5.2, and then use harmonic averaging to obtain face-transmissibilities. That is, for neighboring cells i and j , we compute $T_{ij} = (T_{i,j}^{-1} + T_{j,i}^{-1})^{-1}$ as in (4.51) on page 138.

```
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
```

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

$$\bar{\mathbf{v}}[f] = -\frac{\mathbf{T}[f]}{\mu} (\text{grad}(\mathbf{p}) - g \boldsymbol{\rho}_a[f] \text{grad}(z)), \quad (7.8)$$

where the density at the interface is evaluated using the arithmetic average

$$\boldsymbol{\rho}_a[f] = \frac{1}{2}(\boldsymbol{\rho}[C_1(f)] + \boldsymbol{\rho}[C_2(f)]). \quad (7.9)$$

Similarly, we can write the continuity equation for each cell c as

$$\frac{1}{\Delta t} \left[(\phi(\mathbf{p})[c] \boldsymbol{\rho}(\mathbf{p})[c])^{n+1} - (\phi(\mathbf{p})[c] \boldsymbol{\rho}(\mathbf{p})[c])^n \right] + \operatorname{div}(\boldsymbol{\rho}_a \mathbf{v})[c] = \mathbf{0}. \quad (7.10)$$

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, p_0 is the pressure field at the *previous* time step (i.e., \mathbf{p}^n), while p is the pressure at the *current* time step (\mathbf{p}^{n+1}). Having defined the discrete expression for Darcy-fluxes, we can check that this is in agreement with our initial pressure field by computing the magnitude of the flux, $\operatorname{norm}(v(p_{\text{init}})) * \text{day}$. The result is $1.5 \times 10^{-6} \text{ m}^3/\text{day}$, which should convince us that the initial state of the reservoir is sufficiently close to equilibrium.

7.2.3 Well model

The production well will appear as a source term in the pressure equation. We therefore need to define an expression for flow rate in all cells in which the well is connected to the reservoir (which we will refer to as well connections). Inside the well, we assuming instantaneous flow so that the pressure drop is always hydrostatic. For a horizontal well, the hydrostatic term is zero and could obviously be disregarded, but we include it for completeness and as a robust precaution, in case we later want to reuse the code with a different well path. Approximating the fluid density in the well as constant, computed at bottom-hole pressure, the pressure $\mathbf{p}_c[w]$ in connection w of well $N_w(w)$ is given by

$$\mathbf{p}_c[w] = \mathbf{p}_{bh}[N_w(w)] + g \Delta z[w] \rho(\mathbf{p}_{bh}[N_w(w)]), \quad (7.11)$$

where $\Delta z[w]$ is the vertical distance from the bottom-hole to the connection. We use the standard Peaceman model introduced in Section 4.3.2 to relate the pressure at the well connection to the average pressure inside the grid cell. Using the well-indices provided in \mathbb{W} , the mass flow-rate at connection c is then given by

$$\mathbf{q}_c[w] = \frac{\rho(\mathbf{p}[N_c(w)])}{\mu} \operatorname{WI}[w] (\mathbf{p}_c[w] - \mathbf{p}[N_c(w)]), \quad (7.12)$$

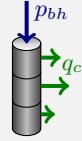
where $\mathbf{p}[N_c(w)]$ is the pressure in the cell $N_c(w)$ containing connection w . In our code, this model is implemented as follows:

```

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 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 by specifying a fixed bottom-hole pressure

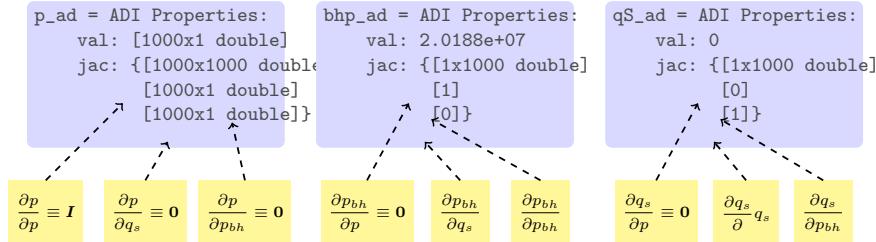
```
ctrlEq = @(bhp) bhp-100*barsa;
```

7.2.4 The simulation loop

What now remains is to set up a simulation loop that will evolve the transient pressure. We start by initializing the AD variables. For clarity, we append `_ad` to all variable names 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 to 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
```

Simulation results from all time steps are stored in a structure `sol`. For efficiency, this structure is preallocated and initialized 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 consists of an outer and an inner loop. The outer loop updates the time step, advances the solution one step forward 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);
```

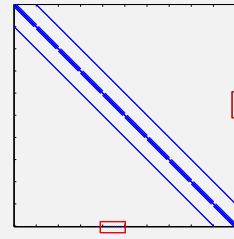
Most of the lines we have implemented so far are fairly standard, except perhaps for the definition of the residual equations as anonymous functions, and equivalent statements can be found in almost any computer program solving this type of time-dependent equation by an implicit method. Now, however, comes what is normally the tricky part: linearization of the equations 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 – *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 in MRST that overloads the elementary operators used to define the residual equations. The calling sequence is obviously more complex than the one depicted in Figure A.6 on page 611, but the operators used are in fact only the three elementary operators plus, minus, and multiply 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 also evaluates the derivatives of each equation with respect to each independent variable and collects 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 gives a Jacobi matrix that has a heptadiagonal structure, except for the 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 7.2 shows a plot of the dynamics of the solution. Initially, the pressure is in hydrostatic equilibrium as shown in Figure 7.1. As the well

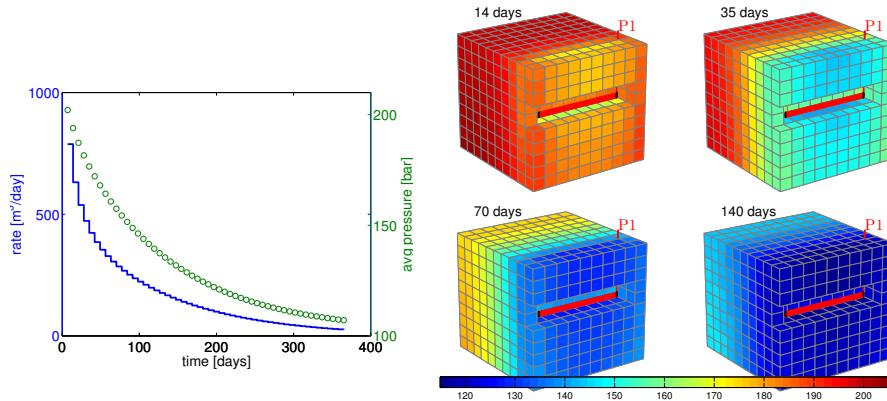


Fig. 7.2. Time evolution of the pressure solution for the compressible single-phase problem. The plot to the left shows the well rate (blue line) and average reservoir pressure (green circles) as function of time, and the plots to the right show the pressure after two, five, ten, and twenty pressure steps.

starts to drain the reservoir, there is a drawdown in the pressure near the well which gradually propagates from the well and outward. As a result, the average pressure inside the reservoir is reduced, which again causes a decay in the production rate.

COMPUTER EXERCISES:

41. Apply the compressible pressure solver introduced above to the quarter five-spot problem discussed in Section 5.4.1.
42. Apply the compressible pressure solver to the three different grid models studied in Section 5.4.3 that were derived from the `seamount` data set. Replace the fixed boundary conditions by a no-flow condition.
43. Use the implementation introduced in Section 7.2 as a template to develop a solver for slightly compressible flow (7.3). More details about this model can be found on page 124 in Section 4.2. 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.
44. Extend the compressible solver developed in this section to incorporate other boundary conditions than no flow.
45. Try to compute time-of-flight by extending the equation set to also include the time-of-flight equation (4.39). Hint: the time-of-flight and the pressure equations need not be solved as a coupled system.
46. Same as above, except that you should try to reuse the solver introduced in Section 5.3. Hint: you must first reconstruct fluxes from the computed pressure and then construct a state object to communicate with the TOF solver.

7.3 Pressure-dependent viscosity

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 discuss two examples that hopefully demonstrate this aspect.

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 illustrate, we 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

$$\mu(p) = \mu_0 [1 + c_\mu(p - p_r)]. \quad (7.13)$$

Having a pressure dependence means that we 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.,

$$\mathbf{q}_c[w] = \frac{\rho(\mathbf{p}[N_c(w)])}{\mu(\mathbf{p}[N_c(w)])} \text{WI}[w] (\mathbf{p}_c[w] - \mathbf{p}[N_c(w)]). \quad (7.14)$$

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,

$$\mathbf{v}[f] = -\frac{\mathbf{T}[f]}{\mu_a[f]} (\text{grad}(\mathbf{p}) - g \rho_a[f] \text{grad}(z)), \quad (7.15)$$

or replace the quotient of the transmissibility and the face viscosity by the harmonic average of the mobility $\lambda = K/\mu$ in the adjacent cells. Both choices introduce changes in the structure of the discrete nonlinear system, but because we are using automatic differentiation, all we have to do is code the corresponding formulas. 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 (changes marked in red):

```
v = @(p) -(T./mu(avg(p))).*( grad(p) - g*avg(rho(p)).*gradz );
```

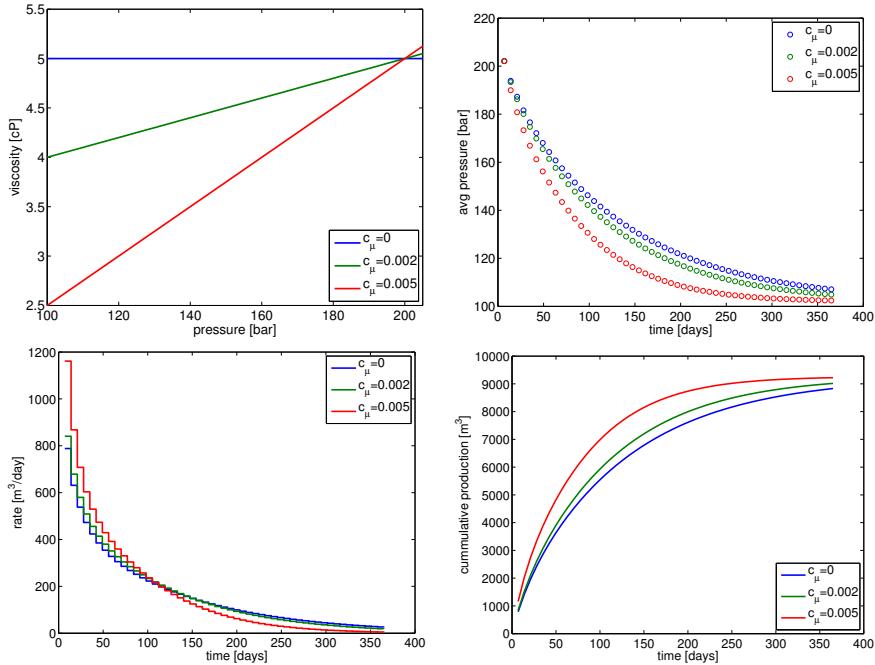


Fig. 7.3. The effect of increasing the degree of pressure-dependence for the viscosity.

and similarly for flow rate through each well connection:

$$q_{\text{conn}} = @(\mathbf{p}, \mathbf{bhp}) \mathbf{WI}.*(\rho(\mathbf{wc}))./\mu(\mathbf{p}(\mathbf{wc})) .* (\mathbf{p}_{\text{conn}}(\mathbf{bhp}) - \mathbf{p}(\mathbf{wc}));$$

In Figure 7.3 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 drawdown 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 drawdown effect of the well will propagate into the reservoir, inducing a reduction in reservoir pressure that eventually will cause production to cease. In terms of overall production, a stronger pressure dependence may be more advantageous as it leads to a higher total recovery and higher cumulative production early in the production period.

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:

$$\mathbf{v}[f] = -\mathbf{A}[f] (\mathbf{grad}(\mathbf{p}) - g \rho_a[f] \mathbf{grad}(z)). \quad (7.16)$$

The face mobility $\Lambda[f]$ can be defined in the same way as the transmissibility is defined in terms of the half transmissibilities using harmonic averages. That is, if $\mathbf{T}[f, c]$ denotes the half transmissibility associated with face f and cell c , the face mobility $\Lambda[f]$ for face f can be written as

$$\Lambda[f] = \left(\frac{\mu[C_1(f)]}{\mathbf{T}[f, C_1(f)]} + \frac{\mu[C_2(f)]}{\mathbf{T}[f, C_2(f)]} \right)^{-1}. \quad (7.17)$$

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,:);
fmob  = @ (mu,p) 1./ (hf2if*(mu(p(hf2cn))./hT));

v = @ (p) -fmob(mu,p).*( grad(p) - g*avg(rho(p)).*gradz );
```

Here, `hf2cn` represents the maps C_1 and C_2 that enable 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 $C_1(f)$ and $C_2(f)$ for each global face f .

Using the harmonic average for a homogeneous model 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} .

COMPUTER EXERCISES:

47. Investigate the claim that the difference between using an arithmetic average of the viscosity and a harmonic average of the fluid mobility is typically small. To this end, you can for instance use the following sub-sample from the SPE 10 data set: `rock = getSPE10rock(41:50,101:110,1:10)`

7.4 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 or 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 mixture, it exhibits properties like a solid and you may be able to run across its surface. However, if you go too slow, the fluid behaves more like a liquid and you 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 tumble around randomly and present large resistance to flow. When the flow velocity increases, the viscosity decreases as the molecules gradually align themselves in the direction of increasing shear rate. A model of the rheology is given by

$$\mu = \mu_\infty + (\mu_0 - \mu_\infty) \left(1 + \left(\frac{K_c}{\mu_0} \right)^{\frac{2}{n-1}} \dot{\gamma}^2 \right)^{\frac{n-1}{2}}, \quad (7.18)$$

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

$$\dot{\gamma}_{\text{app}} = 6 \left(\frac{3n+1}{4n} \right)^{\frac{n}{n-1}} \frac{|\vec{v}|}{\sqrt{K\phi}}. \quad (7.19)$$

Combining (7.18) and (7.19), we can write our model for the viscosity as

$$\mu = \mu_0 \left(1 + \bar{K}_c \frac{|\vec{v}|^2}{K\phi} \right)^{\frac{n-1}{2}}, \quad \bar{K}_c = 36 \left(\frac{K_c}{\mu_0} \right)^{\frac{2}{n-1}} \left(\frac{3n+1}{4n} \right)^{\frac{2n}{n-1}}, \quad (7.20)$$

where we for simplicity have assumed that $\mu_\infty = 0$.

Rapid prototyping

In the following, we 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(eq{1});
upd = -(eq.jac{1} \ eq.val); % Newton update

```

To finish the solver, we 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 increases by almost a factor six since we now 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 implemented for us already: if $\text{div}(\mathbf{x}) = \mathbf{D}' * \mathbf{x}$ computes the discrete divergence in each cell of the field \mathbf{x} defined at the faces, then $\text{cavg}(\mathbf{x}) = 1/6 * \text{abs}(\mathbf{D}') * \mathbf{x}$ computes the average of \mathbf{x} for each cell. In other words, our well equation becomes:

```
wavg = @(eta) 1/6*abs(D(:,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

$$\vec{v}_i = \sum_{j \in N(i)} \frac{v_{ij}}{V_i} (\vec{c}_{ij} - \vec{c}_i), \quad (7.21)$$

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(D), 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));
```

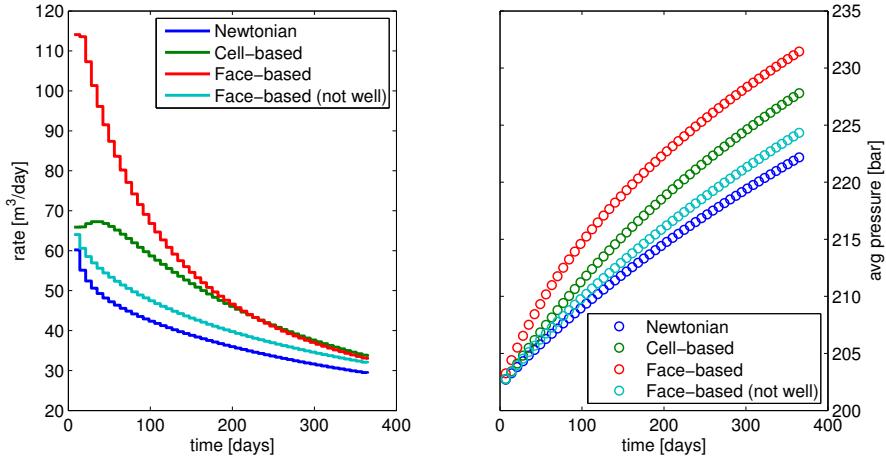


Fig. 7.4. Single-phase injection of a highly viscous, shear-thinning fluid computed by four different simulation methods: (i) fluid assumed to be Newtonian, (ii) shear multiplier η computed in cells, (iii) shear multiplier computed at faces, and (iv) shear multiplier computed at faces, but $\eta \equiv 1$ used in well model.

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 7.4 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 causes a decrease in the viscosity which leads 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

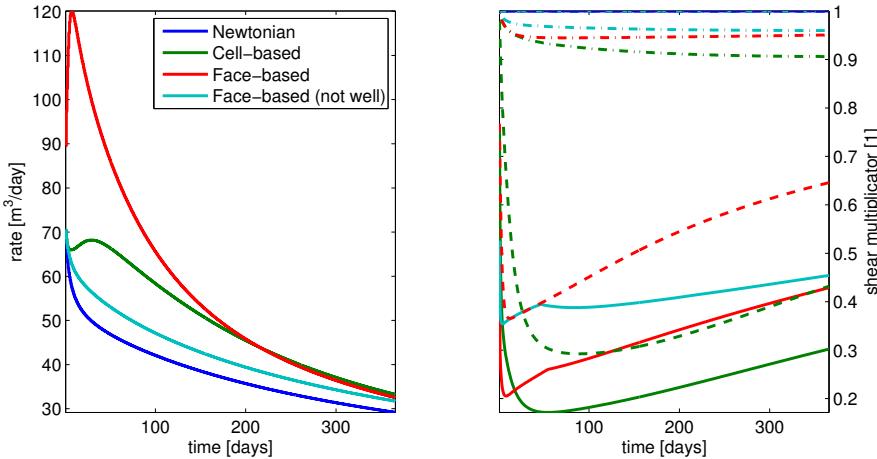


Fig. 7.5. Single-phase injection of a highly viscous, shear-thinning fluid; simulation with $\Delta t = 1/520$ year. The right plot shows the evolution of η as a function of time: solid lines show $\min(\eta)$ over all cells, dashed lines $\min(\eta)$ over the perforated cells, and dash-dotted lines average η value.

Figure 7.5, which also includes plots of the evolution of $\min(\eta)$ as a 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 results in a delay in the peak in the injection rate and a much smaller injected volume.

We 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 still have to understand features and limitations of the models and discretizations you choose to prototype.

COMPUTER EXERCISES:

48. Investigate whether the large differences observed in Figures 7.4 and 7.5 between the cell-based and face-based approaches to the non-Newtonian flow problem is a result of insufficient grid resolution.
49. The non-Newtonian fluid example has 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?

7.5 Thermal effects

As another example of rapid prototyping, we extend the single-phase flow model (7.1) to account for thermal effects. That is, we assume that $\rho(p, T)$ is now a function of pressure and temperature T and extend our model to also include conservation on energy,

$$\frac{\partial}{\partial t}[\phi\rho] + \nabla \cdot [\rho\vec{v}] = q, \quad \vec{v} = -\frac{K}{\mu}[\nabla p - g\rho\nabla z] \quad (7.22a)$$

$$\frac{\partial}{\partial t}[\phi\rho E_f(p, t) + (1 - \phi)E_r] + \nabla \cdot [\rho H_f \vec{v}] - \nabla \cdot [\kappa \nabla T] = q_e \quad (7.22b)$$

Here, the rock and the fluid are assumed to be in local thermal equilibrium. In the energy equation (7.22b), E_f is energy density per mass of the fluid, $H_f = E_f + p/\rho$ is enthalpy density per mass, E_r is energy per volume of the rock, and κ is the heat conduction coefficient of the rock. Fluid pressure p and temperature T are used as primary variables.

As in the original isothermal simulator, we must first define constitutive relationships that express the various physical quantities in terms of the primary variables. The energy equation includes heating of the solid rock, and we therefore start by defining a quantity that keeps track of the solid volume, which also depends on pressure:

```
sv = @p G.cells.volumes - pv(p);
```

For the fluid model, we use

$$\begin{aligned} \rho(p, T) &= \rho_r [1 + \beta_T(p - p_r)] e^{-\alpha(T - T_r)}, \\ \mu(p, T) &= \mu_0 [1 + c_\mu(p - p_r)] e^{-c_T(T - T_r)}, \end{aligned} \quad (7.23)$$

where $\rho_r = 850 \text{ kg/m}^3$ is the density and $\mu_0 = 5 \text{ cP}$ is the viscosity of the fluid at reference conditions with pressure $p_r = 200 \text{ bar}$ and temperature $T_r = 300 \text{ K}$. The constants are $\beta_T = 10^{-3} \text{ bar}^{-1}$, $\alpha = 5 \times 10^{-3} \text{ K}^{-1}$, $c_\mu = 2 \times 10^{-3} \text{ bar}^{-1}$, and $c_T = 10^{-3} \text{ K}^{-1}$. This translates to the following code:

```
mu0 = 5*centi*poise;
cmup = 2e-3/barsa;
cmut = 1e-3;
T_r = 300;
mu = @p,T mu0*(1+cmup*(p-p_r)).*exp(-cmut*(T-T_r));

beta = 1e-3/barsa;
alpha = 5*1e-3;
rho_r = 850*kilogram/meter^3;
rho = @p,T rho_r .* (1+beta*(p-p_r)) .* exp(-alpha*(T-T_r));
```

We use a simple linear relation for the enthalpy, which is based on the thermodynamical relations that give

$$dH_f = c_p dT + \left(\frac{1 - \alpha T_r}{\rho} \right) dp, \quad \alpha = -\frac{1}{\rho} \frac{\partial \rho}{\partial T} \Big|_p, \quad (7.24)$$

where $c_p = 4 \times 10^3$ J/kg. The corresponding code for the enthalpy and energy densities reads:

```
Cp = 4e3;
Hf = @(p,T) Cp*T+(1-T_r*alpha).*(p-p_r)./rho(p,T);
Ef = @(p,T) Hf(p,T) - p./rho(p,T);
Er = @(T) Cp*T;
```

We defer discussing the details of these new relationships and only note that it is important that the thermal potentials E_f and H_f are consistent with the equation-of-state $\rho(p, T)$ to get a physically meaningful model.

Having defined all constitutive relationships in terms of anonymous functions, we can set up the equation for mass conservation and Darcy's law (with transmissibility renamed to T_p to avoid name clash with temperature):

```
v = @(p,T) -(Tp./mu(avg(p),avg(T))).*(grad(p) - avg(rho(p,T)).*gdz);
pEq = @(p,T,p0,T0,dt) ...
    (1/dt)*(pv(p).*rho(p,T) - pv(p0).*rho(p0,T0)) ...
    + div( avg(rho(p,T)).*v(p,T) );
```

The energy equation (7.22b) is a bit more complicated. The accumulation and the heat-conduction terms are on the same form as the operators appearing in (7.22a) and can hence be discretized in the same way. This means that we use a rock object to compute transmissibilities for κ instead of K :

```
tmp = struct('perm',4*ones(G.cells.num,1));
hT = computeTrans(G, tmp);
Th = 1 ./ accumarray(cf, 1 ./ hT, [nf, 1]);
Th = Th(intInx);
```

The remaining term in (7.22b), $\nabla \cdot [\rho H_f \vec{v}]$, represents advection of enthalpy and has a differential operator on the same form as the transport equations discussed in Section 4.4.3 and must hence be discretized by an upwind scheme. To this end, we introduce a new discrete operator that will compute the correct upwind value for the enthalpy density,

$$\text{upw}(\mathbf{H})[f] = \begin{cases} \mathbf{H}[C_1(f)], & \text{if } \mathbf{v}[f] > 0, \\ \mathbf{H}[C_2(f)], & \text{otherwise.} \end{cases} \quad (7.25)$$

With this, we can set up the energy equation on residual form

```
upw = @(x,flag) x(C(:,1)).*double(flag)+x(C(:,2)).*double(~flag);

hEq = @(p, T, p0, T0, dt) ...
    (1/dt)*(pv(p).*rho(p,T).*Ef(p,T) + sv(p).*Er(T) ...) ...
    - pv(p0).*rho(p0,T0).*Ef(p0,T0) - sv(p0).*Er(T0) ...
    + div( upw(Hf(p,T),v(p,T)>0).*avg(rho(p,T)).*v(p,T) ...) ...
    + div( -Th.*grad(T));
```

and are thus almost done. As a last technical detail, we must also make sure that the energy transfer in injection and production wells is modelled correctly using appropriate upwind values:

```

qw      = q_corn(p_ad, T_ad, bhp_ad);
eq1     = pEq(p_ad, T_ad, p0, T0, dt);
eq1(wc) = eq1(wc) - qw;
hq      = Hf(bhp_ad,bhT).*qw;
Hcells  = Hf(p_ad,T_ad);
hq(qw<0) = Hcells(wc(qw<0)).*qw(qw<0);
eq2     = hEq(p_ad,T_ad, p0, T0,dt);
eq2(wc) = eq2(wc) - hq;

```

Here, we evaluate the enthalpy using *cell values* for pressure and temperature for production wells (for which $qw < 0$) and pressure and temperatures at the *bottom hole* for injection wells.

What remains, are trivial changes to the iteration loop to declare the correct variables as AD structures, evaluate the discrete equations, collect their residuals, and update the state variables. These details can be found in the complete code given in `singlePhaseThermal.m` and have been left out for brevity.

Understanding thermal expansion

Except for the modifications discussed above, the setup is the exact same as in Section 7.2. That is, the reservoir is a $200 \times 200 \times 50$ m³ rectangular box with homogeneous permeability of 30 mD, constant porosity 0.3, and a rock compressibility of 10^{-6} bar⁻¹, realized on a $10 \times 10 \times 10$ Cartesian grid. The reservoir is realized on a $10 \times 10 \times 10$ Cartesian grid. Fluid is drained from a horizontal well perforated in cells with indices $i = 2, j = 2, \dots, 9$, and $k = 5$, and operating at a constant bottom-hole pressure of 100 bar. Initially, the reservoir has constant temperature of 300 K and is in hydrostatic equilibrium with a datum pressure of 200 bar specified in the uppermost cell centroids.

In the same way as in the isothermal case, the open well will create a pressure drawdown that propagates into the reservoir. As more fluid is produced from the reservoir, the pressure will gradually decay towards a steady state with pressure values between 101.2 and 104.7 bar. Figure 7.6 shows that the simulation predicts a faster pressure drawdown, and hence a faster decay in production rates, if thermal effects are taken into account.

The change in temperature of an expanding fluid will not only depend on the initial and final pressure, but also on the type of process in which the temperature is changed:

- In a *free expansion*, the internal energy is preserved and the fluid does no work. That is, the process can be described by the following differential:

$$\frac{dE_f}{dp} \Delta p + \frac{dE_f}{dT} \Delta T = 0. \quad (7.26)$$

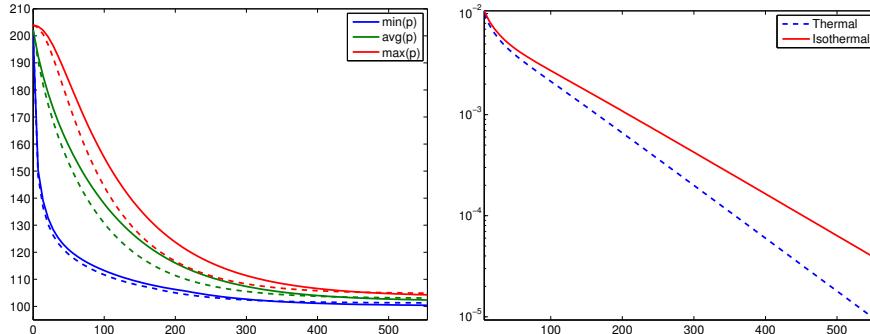


Fig. 7.6. To the left, time evolution for pressure for an isothermal simulation (solid lines) and a thermal simulation with $\alpha = 5 \cdot 10^{-3}$ (dashed lines). To the right, decay in production rate at the surface.

When the fluid is an ideal gas, the temperature is constant, but otherwise the temperature will either increase or decrease during the process depending on the initial temperature and pressure.

- In a reversible process, the fluid is in thermodynamical equilibrium and does positive work while the temperature decreases. The linearized function associated with this *adiabatic expansion* reads,

$$dE + \frac{p}{\rho V} dV = dE + p d\left(\frac{1}{\rho}\right) = 0. \quad (7.27)$$

- In a Joule–Thomson process, the enthalpy remains constant while the fluid flows from higher to lower pressure under steady-state conditions and without change in kinetic energy. That is,

$$\frac{dH_f}{dp} \Delta p + \frac{dH_f}{dT} \Delta T = 0. \quad (7.28)$$

Our case is a combination of these three processes and their interplay will vary with the initial temperature and pressure as well as with the constants in the fluid model for $\rho(p, T)$. To better understand a specific case, we can use (7.26) to (7.28) to compute the temperature change that would take place for an observed pressure drawdown if only one of the processes took place. Computing such linearized responses for thermodynamical functions is particularly simple using automatic differentiation. Assuming we know the reference state (p_r, T_r) at which the process starts and the pressure p_e after the process has taken place, we initialize the AD variables and compute the pressure difference:

```
[p,T] = initVariablesADI(p_r,T_r);
dp     = p_e - p_r;
```

Then we can solve (7.26) or (7.28) for ΔT and use the result to compute the temperature change resulting from a free expansion or a Joule–Thomson expansion:

```

E     = Ef(p,T);
dEdp = E.jac{1};
dEdT = E.jac{2};
Tfr  = T_r - dEdp*dp/dEdT;

hf   = Hf(p,T);
dHdp = hf.jac{1};
dHdT = hf.jac{2};
Tjt  = T_r - dHdp*dp/dHdT;

```

The temperature change after a reversible (adiabatic) expansion is not described by a total differential. In this case we have to specify that p should be kept constant. This is done by replacing the AD variable p by an ordinary variable `double(p)` in the code at the specific places where p appears in front of a differential, see (7.27).

```

E   = Ef(p,T) + double(p)./rho(p,T);
dEdp = hf.jac{1};
dEdT = hf.jac{2};
Tab = T_r - dEdp*dp/dEdT;

```

The same kind of manipulation can be used to study alternative linearizations of systems of nonlinear equations and the influence of neglecting some of the derivatives when forming Jacobians.

To illustrate how the interplay between the three processes can change significantly and lead to quite different temperature behavior, we will compare the predicted evolution of the temperature field for $\alpha = 5 \times 10^{-n}$, $n = 3, 4$, as shown in Figures 7.7 and 7.8. The change in behavior between the two figures is associated with the change in sign of $\partial E / \partial p$,

$$dE = \left(c_p - \frac{\alpha T}{\rho} \right) dT + \left(\frac{\beta_T p - \alpha T}{\rho} \right) dp, \quad \beta_T = \frac{1}{\rho} \frac{\partial \rho}{\partial p} \Big|_T. \quad (7.29)$$

In the isothermal case and for $\alpha = 5 \times 10^{-4}$, we have that $\alpha T < \beta_T p$ so that $\partial E / \partial p > 0$. The expansion and flow of fluid will cause an instant heating near the well-bore, which is what we see in the initial temperature increase for the maximum value in Figure 7.7. The Joule–Thomson coefficient $(\alpha T - 1)/(c_p \rho)$ is also negative, which means that the fluid gets heated if it flows from high pressure to low pressure in a steady-state flow. This is seen by observing the temperature in the well perforations. The fast pressure drop in these cells causes an almost instant cooling effect, but soon after we see a transition in which most of the cells with a well perforation start having the highest temperature in the reservoir because of heating from the moving fluids. For $\alpha = 5 \times 10^{-3}$, we have that $\alpha T > \beta_T p$ so that $\partial E / \partial p < 0$ and likewise the Joule–Thomson coefficient is positive. The moving fluids will induce a cooling effect and hence the minimum temperature is observed at the well for a longer time. The weak kink in the minimum temperature curve is the result of the

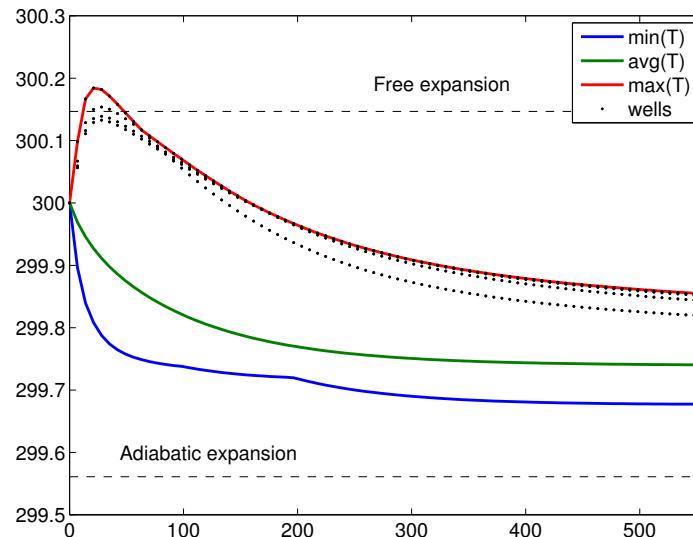
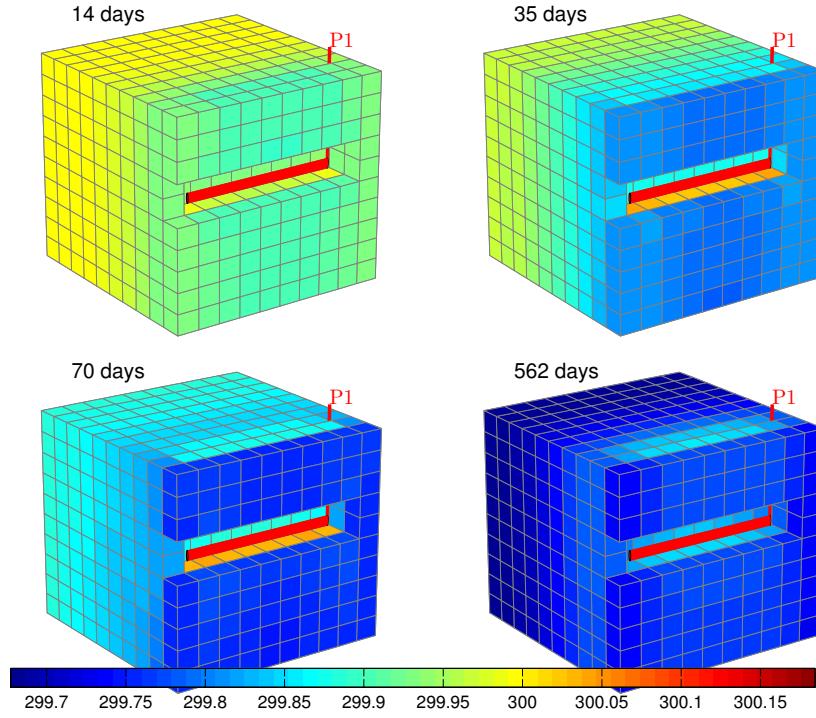


Fig. 7.7. Time evolution of temperature for a compressible, single-phase problem with $\alpha = 5 \cdot 10^{-4}$. The upper plots show four snapshots of the temperature field. The lower plot shows minimum, average, maximum, and well-perforation values.

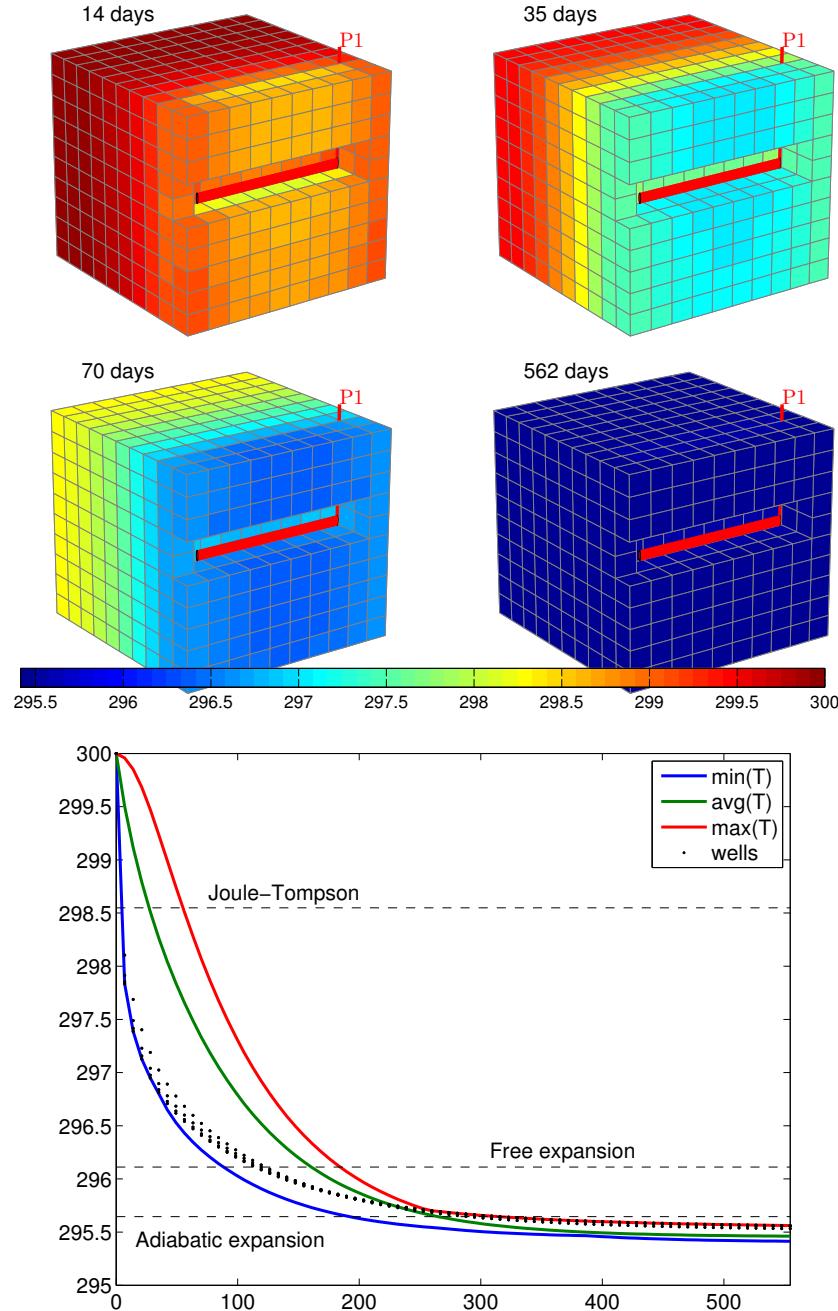


Fig. 7.8. Time evolution of temperature for a compressible, single-phase problem with $\alpha = 5 \cdot 10^{-3}$. The upper plots show four snapshots of the temperature field. The lower plot shows minimum, average, maximum, and well-perforation values.

point of minimum temperature moving from being at the bottom front side to the far back of the reservoir. The cell with lowest temperature is where the fluid has done most work, neglecting heat conduction. In the beginning this is the cell near the well since the pressure drop is largest there. Later it will be the cell furthest from the well since this is where the fluid can expand most.

Computational performance

The observant reader may have realized that the code presented above contains a number of redundant function evaluations that may potentially add significantly to the overall computational cost: In each nonlinear iteration we keep re-evaluating quantities that depend on p_0 and T_0 even though these stay constant for each time step. This can easily be avoided by moving the definition of the anonymous functions evaluating the residual equations inside the outer time loop. The main contribution to potential computational overhead, however, comes from repeated evaluations of fluid viscosity and density. Because each residual equation is defined as an anonymous function, $v(p, T)$ appears three times for each residual evaluation, once in pEq and twice in hEq . This, in turn, translates to three calls to $\mu(\text{avg}(p), \text{avg}(T))$ and *seven* calls to $\rho(p, T)$, and so on. In practice, the number of actual function evaluations is smaller since the MATLAB interpreter most likely has some kind of built-in intelligence to spot and reduce redundant function evaluations. Nonetheless, to cure this problem, we can move the computations of residuals inside a function so that the constitutive relationships can be computed one by one and stored in temporary variables. The disadvantage is that we increase the complexity of the code and move one step away from the mathematical formulas describing the method. This type of optimization should therefore only be introduced after the code has been profiled and redundant function evaluations have proved to have a significant computational cost.

COMPUTER EXERCISES:

50. Perform a more systematic investigation of how changes in α affect the temperature and pressure behavior. To this end, you should change α systematically, e.g., from 0 to 10^{-2} . What is the effect of changing β , the parameters c_μ and c_T for the viscosity, or c_p in the definition of enthalpy?
51. Use the MATLAB profiling tool to investigate to what extent the use of nested anonymous functions causes redundant function evaluations or introduces other types of computational overhead. Hint: to profile the CPU usage, you can use the following call sequence

```
profile on, singlePhaseThermal; profile off; profile report
```

Try to modify the code as suggested above to reduce the CPU time. How low can you get the ratio between the cost of constructing the linearized system and the cost of solving it?

Part III

Multiphase Flow

Page: 234

job: mrst-book

macro: svmono.cls

date/time: 19-Oct-2017/1:29

Mathematical Models for Multiphase Flow

Up to now, we have only considered flow of a single fluid phase. For most applications of reservoir simulation, however, one is interested in modelling how one fluid phase displaces one or more other fluid phases. In petroleum recovery, typical examples will be water or gas flooding in which injected water or gas displaces the resident hydrocarbon phase(s). Likewise, in geological carbon sequestration, the injected CO₂ forms a supercritical fluid phase that displaces the resident brine. In both cases, multiple phases will flow simultaneously throughout the porous medium when viewed from the scale of a representative elementary volume, even if the fluids are immiscible and do not mix on the microscale.

To model this flow, we will introduce three new physical properties of multiphase models (saturation, relative permeability, and capillary pressure) and discuss how one can use these to extend Darcy's law to multiphase flow and combine it with conservation of mass for each fluid phase to develop a model that describes multiphase displacements. The resulting system of partial differential equations is parabolic in the general case, but has a mixed elliptic-hyperbolic mathematical character, in which fluid pressures tend to behave as following a near elliptic equation, while the transport of fluid phases has a strong hyperbolic character. It is therefore common to use a so-called fractional flow formulation to write the flow equations as a coupled system consisting of a pressure equation describing the evolution of one of the fluid pressures and one or more saturation equations that describe the transport of the fluid phases. The mixed elliptic-hyperbolic nature is particularly evident in the case of immiscible, incompressible flow, in which case the pressure equation simplifies to a Poisson equation on the same form as we have studied in the previous chapters.

Fractional-flow formulations are very popular among mathematicians and computer scientists who develop and analyze numerical methods since these formulations reveal the mathematical character of different parts of the equation system explicitly. Fractional-flow formulations naturally lead to operator splitting methods in which the pressure and transport equations are

solved in separate steps, potentially using specialized numerical methods that are developed to utilize special characteristics of each sub-equation. Examples include the classical IMPES (implicit pressure, explicit saturation) method as well as streamline simulation [69] and recent multiscale methods [255, 177, 98, 179, 180]. In industry, on the other hand, the most widespread approach is to use a compressible description (even for near-incompressible flows) and solve the coupled flow equations using a fully-implicit discretization. Nonetheless, also in this case the mixed elliptic-hyperbolic character of the model equations plays a key role in developing efficient preconditioning strategies for the linearized system.

In this chapter we will introduce new physical effects that appear in multiphase flow, discuss general multiphase models in some detail, derive the fractional-flow formulation in the special case of immiscible flow, and analyze the mathematical character of the system in various limiting cases. Chapter 9 introduces various methods for solving hyperbolic transport equation and reviews some of the supporting theory. Then, in Chapter 10, we focus entirely on the incompressible case and show how we can easily reuse the elliptic solvers developed in the previous chapters and combine them with a set of simple first-order transport solvers that are implemented in the `incomp` module of MRST. To simplify the discussion, this and the next two chapters will mainly focus on two-phase, immiscible systems, but the most crucial equations will be stated and developed for the general multiphase case. In Chapter 11, we return to the general case and discuss the compressible models and numerical methods that are used for three-phase flow in most contemporary commercial simulators.

8.1 New physical properties and phenomena

As we have seen previously, a Darcy-type continuum description of a reservoir fluid system means that any physical quantity defined at a point \vec{x} represents an average over a representative elementary volume (REV). Let us consider a system with two or more fluid phases that are immiscible so that no mass transfer takes place between the different phases. This means that the fluid phases will not mix and form a solution on the microscale but rather stay as separate volumes or layers separated by a curved meniscus as illustrated in Figure 8.1. However, when considering the flow averaged over an REV, the fluid phases will generally not be separated by a sharp interface so that two or more phases may occupy the same point in the continuum description. In this section we will introduce the new fundamental concepts that are necessary to understand multiphase flow and formulate continuum models that describe the simultaneous flow of two or more fluid phases taking place at the same point in a reservoir. Unless stated otherwise, the two fluids are assumed to be oil and water when we discuss two-phase systems.

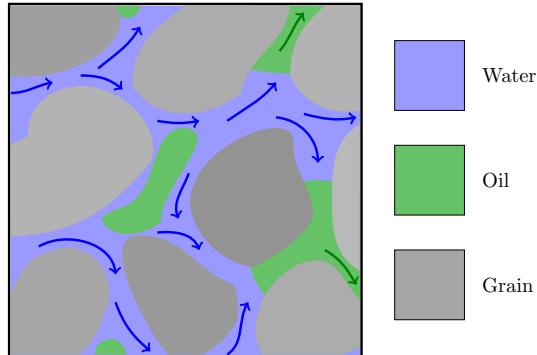


Fig. 8.1. Averaging of a multiphase flow over a representative elementary volume (REV) to obtain a Darcy-scale continuum model.

8.1.1 Saturation

The saturation S_α is defined as the fraction of the pore volume occupied by phase α . In the single-phase models discussed in previous chapters, we assumed that the void space between the solid particles that make up the porous medium was completely filled by fluid. Similarly, for multiphase models we assume that the void space is completely filled with one or more fluid phases, so that

$$\sum_{\alpha} S_{\alpha} = 1. \quad (8.1)$$

In reservoir simulation, it is most common to consider three phases: an aqueous phase (w), an oleic phase (o), and a gaseous (g) phase. Each saturation can vary from 0, which means that the phase is not present at all at this point in space, to 1, which means that the phase fills the complete local pore volume. In most practical cases, however, the range of variability is smaller. Let us for instance consider a rock that was originally deposited in an aqueous environment. During deposition, the pores between rock particles will be completely filled with water. Later, as hydrocarbons start to migrate into what is to become our reservoir, the water will be displaced and the saturation gradually reduced to some small value, typically 5–40%, at which the water can no longer flow and exists as small drops trapped between mineral particles or encapsulated by the invading hydrocarbon phases. The saturation at which water goes from being mobile (funicular state) to being immobile (pendular state) is called the *irreducible water saturation* and is usually denoted S_{wir} or S_{wr} . The irreducible water saturation is determined by the topology of the pore space and the water's affinity to wet the mineral particles relative to that of the invading hydrocarbons, which in turn is determined by the chemical composition of the fluids and the mineral particles. In petroleum literature, one also talks of the *connate water saturation*, usually denoted S_{wc} , which is

the water saturation that exists upon discovery of the reservoir. The quantities S_{wir} and S_{wc} may or may not coincide, but should not be confused. Sometimes, one also sees the notation S_{wi} which may refer to any of the two. In a similar way, if water is later injected to displace the oil, it is generally not possible to flush out all the oil and part of the pore space will be occupied with isolated oil droplets, as illustrated in Figure 8.1. The corresponding residual oil saturation is denoted as S_{or} . In most systems, the water has a larger affinity to wet the rock, which means that S_{or} is usually higher than S_{wr} and S_{wc} : typically are in the range 10–50%.

In many models, each phase may also contain one or more *components*. These may be unique hydrocarbon species like methane, ethane, propane, etc., or other chemical species like polymers, salts, surfactants, tracers, etc. Since the number of hydrocarbon components can be quite large, it is common to group components into pseudo-components. Due to the varying and extreme conditions in a reservoir, the composition of the different phases may change throughout a simulation (and may sometimes be difficult to determine uniquely). To account for this, we therefore need to describe this composition. There are several ways to do this. Herein, we will use the mass fraction of component ℓ in phase α , denoted by c_α^ℓ and defined as

$$c_\alpha^\ell = \frac{\rho_\alpha^\ell}{\rho_\alpha} \quad (8.2)$$

where ρ_α denotes the bulk density of phase α and ρ_α^ℓ the effective density of component ℓ in phase α . In each of phase, the mass fractions should add up to unity, so that for M different components in a system consisting of an aqueous, a gaseous, and an oleic phase, we have:

$$\sum_{\ell=1}^M c_g^\ell = \sum_{\ell=1}^M c_o^\ell = \sum_{\ell=1}^M c_w^\ell = 1. \quad (8.3)$$

We will return to models having three phases and more than one component per phase later in the book. For now, however, we assume that our system consists of two immiscible phases.

8.1.2 Wettability

At the microscale, which is significantly larger than the molecular scale, immiscible fluid phases will be separated by a well-defined, infinitely thin interface. Because cohesion forces between molecules are different on opposite sides, this interface has an associated surface tension (or surface energy), which measures the forces the interface must overcome to change its shape. In the absence of external forces, minimization of surface energy will cause the interface of a droplet of one phase contained within another phase to assume a spherical shape. The interface tension will keep the fluids apart, regardless of the size of the droplet.

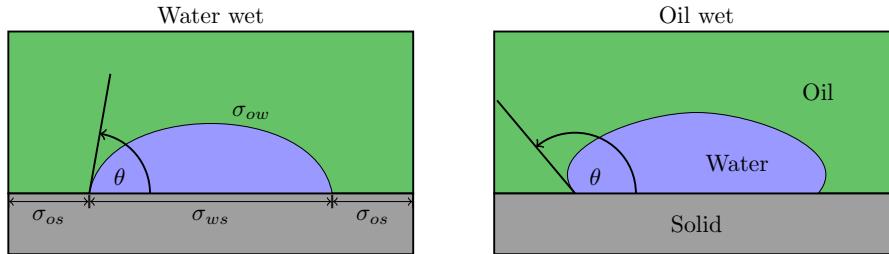


Fig. 8.2. Contact angle θ and surface tension σ for two different oil-water systems. In water-wet systems, $0 \leq \theta < 90^\circ$, whereas $90^\circ < \theta < 180^\circ$ in oil-wet systems.

The microscale flow of our oil-water system will be strongly affected by how the phases attach to the interface of the solid rock. The ability of a liquid phase to maintain contact with a solid surface is called *wettability* and is determined by intermolecular interactions when the liquid and solid are brought together. Adhesive forces between different molecules in the liquid phase and the solid rock will cause liquid droplets to spread across the mineral surface. Likewise, cohesive forces between similar molecules within the liquid phases will cause the droplets to avoid contact with the surface and ball up. When two fluid phases are present in the same pore space, one phase will be more attracted to the mineral particles than the other phase. The preferential phase is referred to as the *wetting phase*, while the other is called the *non-wetting phase*. The balance of the adhesive and cohesive forces determines the contact angle θ shown in Figure 8.2, which is a measure of the wettability of a system that can be related to the interface energies by Young's equation:

$$\sigma_{ow} \cos \theta = \sigma_{os} - \sigma_{ws} \quad (8.4)$$

where σ_{ow} is the interface energy of the oil-water interface and σ_{os} and σ_{ws} are the energies of the oil-solid and water-solid interfaces, respectively. Hydrophilic or water-wet porous media, in which the water shows a greater affinity than oil to stick to the rock surface, are more widespread in nature than hydrophobic or oil-wet media. This explains why S_{or} usually is larger than S_{wr} . In a perfectly water-wetting system, $\theta = 0$ so that water spreads evenly over the whole surface of the mineral grains. Likewise, in a perfectly oil-wet system, $\theta = 180^\circ$ so that water forms spherical droplets at the solid surface.

8.1.3 Capillary pressure

Because of the surface tension, the equilibrium pressure in two phases separated by a curved interface will generally be different. The difference in phase pressures is called the *capillary pressure*:

$$p_c = p_n - p_w, \quad (8.5)$$

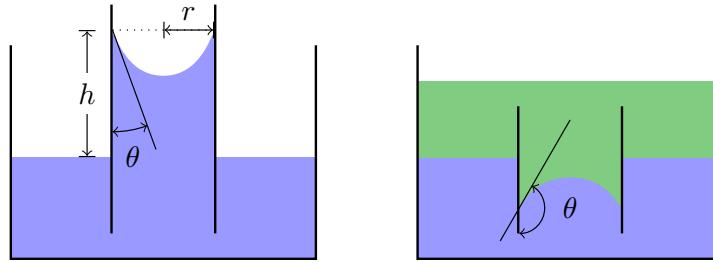


Fig. 8.3. Capillary tubes for a wetting liquid and air (left) and a wetting and a non-wetting liquid (right). (Width of capillary tube is exaggerated).

which will always be positive because the pressure in the non-wetting fluid is higher than the pressure in the wetting fluid. For a water-wet reservoir, the capillary pressure is therefore defined as $p_{cow} = p_o - p_w$, whereas one usually defines $p_{cog} = p_g - p_o$ in a oil-gas system where oil is the wetting phase.

The action of capillary pressures can cause liquids to move in narrow spaces, devoid of or in opposition to other external forces such as gravity. To illustrate this, we consider a thin tube immersed in a wetting and a non-wetting fluid as shown in Figure 8.3. For the wetting fluid, adhesive forces between the solid tube and the liquid will form a concave meniscus and pull the liquid upward against the gravity force. It is exactly the same effect that causes water to be drawn up into a piece of cloth or paper that is dipped into water. In the non-wetting case, the intermolecular cohesion forces within the liquid exceed the adhesion forces between the liquid and the solid, so that a convex meniscus is formed and drawn downwards relative to the liquid level outside of the tube. At equilibrium inside the capillary tube, the upward and the downward forces must balance each other. The force acting upward equals

$$2\pi r(\sigma_{as} - \sigma_{ls}) = 2\pi r \sigma \cos \theta$$

where subscripts a and l refer to air and liquid, respectively. The capillary pressure is defined as force per unit area, or in other words,

$$p_c = \frac{2\pi r \sigma \cos \theta}{\pi r^2} = \frac{2\sigma \cos \theta}{r} \quad (8.6)$$

The force acting downward can be deducted from Archimedes' principle as $\pi r^2 g h (\rho_l - \rho_a)$ and by equating this with the action of the capillary pressure, we obtain

$$p_c = \frac{\pi r^2 g h (\rho_l - \rho_a)}{\pi r^2} = \Delta \rho g h. \quad (8.7)$$

The void space inside a reservoir will contain a large number of narrow pore throats that can be thought of as a bundle of non-connecting capillary tubes of different radius. As we can see from the formulas developed above,

the capillary pressure increases with decreasing tube radius for a fixed interface energy difference between two immiscible fluids. Because the pore size is usually so small, the capillary pressure will play a major role in establishing the fluid distribution inside the reservoir. To see this, consider a hydrocarbon phase migrating upward by buoyancy forces into a porous rock filled with water. If the hydrocarbon phase is to enter the void space in the rock, its buoyancy force must exceed a certain minimum capillary pressure. The capillary pressure that is required to force the first droplet of oil into the rock is called the *entry pressure*. If we consider the rock as a complex assortment of capillary tubes, the first droplet will enter the widest tube which according to (8.6) has the lowest capillary pressure. As the pressure difference between the buoyant oil and the resident water increases, the oil will be able to enter increasingly smaller pore throats and hence reduce the water saturation. This means that there will be a relation between saturation and capillary pressure,

$$p_{cnw} = p_n - p_w = P_c(S_w) \quad (8.8)$$

e.g., as illustrated in Figure 8.4. The slope of the curve is determined by the variability of the pore sizes. If all pores are of similar size, they will all be invaded quickly once we exceed the entry pressure and the curve will be relatively flat so that saturation decays rapidly with increasing capillary pressure. If the pores vary a lot in size, the decrease in saturation with increasing capillary pressure will be more gradual. As for the vertical distribution of fluids, we see that once the fluids have reached a hydrostatic equilibrium, the difference in densities between water and oil dictates the difference in phase pressures and hence the oil saturation increase in the upward direction, which is also illustrated in the figure.

The argument above was developed for the case that an invading non-wetting fluid displaces a wetting fluid. This type of displacement is called *drainage* to signify that the saturation of the wetting phase is decreasing in this type of displacement. The opposite case, called *imbibition*, occurs when a wetting fluid displaces a non-wetting fluid. As an example, let us assume that we inject water to flush out the oil in a pristine reservoir having connate water equal the irreducible water saturation. During this displacement, the water saturation will gradually increase as more water is injected. Hence, the oil saturation decrease until we reach the residual oil saturation at which there is only immobile oil left. The displacement will generally not follow the same capillary curve as the primary drainage curve, as shown in the right plot in Figure 8.4. Likewise, if another drainage displacement takes place starting from S_{or} or a larger oil saturation, the process will generally not follow the imbibition curve. The result is an example of what is called *hysteresis*, in which the behavior of a system depends both on the current state and its previous history. The hysteretic behavior can be explained by pore-scale trapping of oil droplets, by variations in the wetting angle between advancing and receding fluid at the solid interface and by the fact that whereas the drainage process

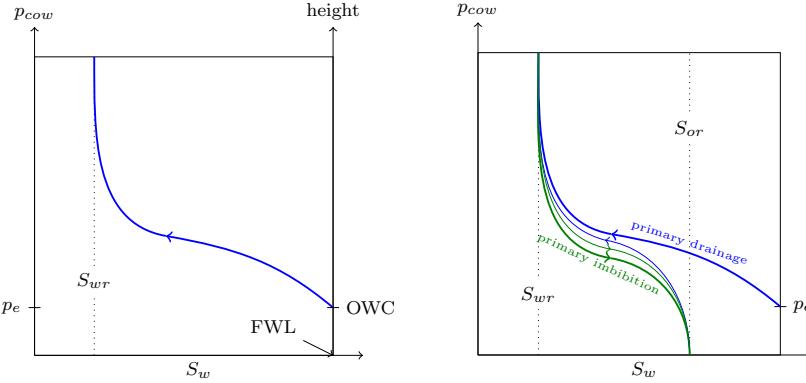


Fig. 8.4. The left plot shows a capillary pressure curve giving the relation between capillary pressure p_{cow} and water saturation S_w . In addition we have included how the capillary pressure and water saturation relate to the height above the free water level (FWL) for a system in hydrostatic equilibrium. Notice that the entry pressure p_e occurs at the oil-water contact (OWC), which found above the free water level for systems with nonzero entry pressures. The right plot shows hysteretic behavior for repeated drainage and imbibition displacements.

is controlled by the size of the widest non-invaded pore throat, the imbibition process is controlled by the size of the narrowest non-invaded pore.

As we can see from Figure 8.4, a relatively large fraction of the oil will be left behind in an immobile state after waterflooding. Several methods for enhanced oil recovery have been developed to mobilize this immobile oil, e.g., by injecting another fluid (e.g., CO₂ or gas) that mixes with immobile oil droplets so that a larger fraction of the oil can be washed out along with the invading fluid. In chemical and microbial methods, one adds chemical substances or small microorganisms to the injected fluids that alter the wetting properties inside the pores. Simulating these processes, however, will require more advanced models than what is considered in the current chapter.

To use the relation between capillary pressure and saturation in practical modeling, it is convenient to express $P_c(S_w)$ as an analytical or tabulated function. In the petroleum industry, this is typically done by using experiments on core samples from the reservoir to develop an empirical model based on observations of the relationship between average p_c and S_w values inside the core models. Each core sample will naturally generate a different capillary curve because of differences in pore-size distribution, porosity, and permeability. To normalize the measured data, it is common to use a so-called Leverett J-function [151], which takes the following form

$$J(S_w) = \frac{P_c}{\sigma \cos \theta} \sqrt{\frac{K}{\phi}}, \quad (8.9)$$

where the surface tension σ and the contact angle θ are measured in the laboratory and are specific to a particular rock and fluid system. The scaling factor $\sqrt{K/\phi}$ is proportional to the characteristic, effective pore-throat radius. The function J can now be obtained as a (tabulated) function of S_w by fitting rescaled observed data (p_c and s_w) to a strictly monotone J -shaped function. Then, the resulting function is used to extrapolate capillary pressure data measured for a given rock to rocks that are similar, but have different permeability, porosity, and wetting properties. One cannot expect to find a J function that is generally applicable since the parameters that affect capillary pressure have very large variations with rock type. Nevertheless, experience has shown that the J -curves correlate well for a given rock type and in reservoir models it is therefore common to derive a J -curve for each specific rock type (facies) that is represented in the underlying geological model.

To set up a multiphase flow simulation, one needs to know the initial saturation distribution inside the reservoir. If we know the location of the oil-water contact, we can estimate the saturation higher up in the formation by combining (8.7) and (8.9)

$$S_w = J^{-1} \left(\frac{\Delta \rho g h}{\sigma \cos \theta} \sqrt{\frac{K}{\phi}} \right).$$

In other application areas than petroleum recovery, it is common to use models that express the capillary pressure directly as an analytical function of the normalized water saturation,

$$\hat{S}_w = \frac{S_w - S_w^{\min}}{S_w^{\max} - S_w^{\min}} \quad (8.10)$$

where S_w^{\max} and S_w^{\min} are the maximum and minimum values the saturation can attain during the displacement. For the primary drainage shown in the left plot of Figure 8.4, it is natural to set $S_w^{\max} = 1$ and $S_w^{\min} = S_{wr}$, whereas $S_w^{\max} = 1 - S_{or}$ and $S_w^{\min} = S_{wr}$ for all subsequent displacements. The following model was proposed by Brooks and Corey [48] to model the relationship between capillary pressure and water saturation in partially saturated media (i.e., in the vadose zone where the two-phase flow consists of water and air, see Section 8.3.5):

$$\hat{S}_w = \begin{cases} (p_c/p_e)^{-n_b}, & \text{if } p_c > p_e \\ 1, & p_c \leq p_e, \end{cases} \quad (8.11)$$

where p_e is the entry pressure of air and $n_b \in [0.2, 5]$ is a parameter related to the pore-size distribution. Another classical model is the one proposed by van Genuchten [249]:

$$\hat{S}_w = \left(1 + (\beta_g p_c)^{n_g} \right)^{-m_g}, \quad (8.12)$$

where β_g is a scaling parameter related to the average size of pores and the exponents n_g and m_g are related to the pore-size distribution.

8.1.4 Relative permeability

In our discussion of incompressible single-phase flow we have seen that the only petrophysical parameter that affects how fast a fluid flows through a porous medium is the absolute permeability K that measures the capacity of the rock to transmit fluids, or alternatively the resistance the rock offers to flow. As described in Chapters 2 and 4, absolute permeability is an intrinsic property of the rock and does not depend on the type of fluid that flows through the rock. In reality, this is not true, mainly because of microscale interactions between the rock fluid that may cause particles to move, pore spaces to be plugged, clays to swell when brought in contact with water, etc. Likewise, liquids and gases may not necessarily experience the same permeability because gas does not adhere to the mineral surfaces in the same way as liquids do. This means that while the flow of liquids is subject to no-slip boundary conditions, gases may experience slippage which gives a pressure-dependent apparent permeability that at low flow rates is higher than the permeability experienced by liquids. This is called the Klinkenberg effect and plays a substantial role for gas flows in low-permeable unconventional reservoirs such as such as coal seams, tight sands, and shale formations. Herein, we will not consider reservoirs where these effects are pronounced and henceforth, we assume, as for the incompressible, single-phase flow models in Chapter 4 that the absolute permeability K is an intrinsic quantity.

When more than one phase is present in the pore space, each phase α will experience an effective permeability K_α^e that is less than the absolute permeability K . Looking at the conceptual drawing Figure 8.1, it is easy to see why this is so. The presence of another phase will effectively present additional 'obstacles', whose interfacial tension offer resistance to flow. Because interfacial tension exists between all immiscible phases, the sum of all the effective phase permeabilities will generally be less than one, i.e.,

$$\sum_{\alpha} K_{\alpha}^e < K.$$

To model this reduced permeability, we introduce a property called *relative permeability* [185], which for an isotropic medium is defined as

$$k_{r\alpha} = K_{\alpha}^e / K. \quad (8.13)$$

Because the effective permeability is always less or equal to the absolute permeability, $k_{r\alpha}$ will take values in the interval between 0 and 1 and $\sum_{\alpha} k_{r\alpha} \leq 1$. For anisotropic media, the relationship between the effective and absolute permeability may in principle be different for each component of the tensors. However, it is still common to define the relative permeability as a scalar quantity that is postulated to be in the following form

$$K_{\alpha}^e = k_{r\alpha} K. \quad (8.14)$$

The relative permeabilities will generally be functions of saturation, which means that we for a two-phase system can write

$$k_{rn} = k_{rn}(S_n) \quad \text{and} \quad k_{rw} = k_{rw}(S_w).$$

It is important to note that the relative permeabilities generally are nonlinear functions of the saturations, so that the sum of the relative permeabilities at a specific location (with a specific composition) is not necessarily equal to one. As for the relationship between saturation and capillary pressure, the relative permeabilities are strongly dependent on the lithofacies and is therefore common practice to associate an unique pair of curves with each rock type represented in the geological model. The relative permeabilities may depend on pore-size distribution, fluid viscosity, and temperature, but these factors are usually ignored in models of conventional reservoirs.

In simplified models, it is common to assume that $k_{r\alpha}$ are monotone functions that assume unique values in $[0, 1]$ for all values $S_\alpha \in [0, 1]$, so that $k_{r\alpha} = 1$ corresponds to the case with fluid α occupying the entire pore space and $k_{r\alpha} = 0$ when $S_\alpha = 0$; see the left plot in Figure 8.5. In practice, $k_{r\alpha} = 0$ occurs when the fluid phase becomes immobilized for $S_\alpha \leq S_{\alpha r}$, giving relative permeability curves as shown in the right plot in Figure 8.5. The preferential wettability can be deducted from the point where two curves cross. Here, the curves cross for $S_w > 0$, which indicates that the system is water-wet.

Going back to our previous discussion of hysteresis, we will generally expect that the relative permeability curves are different during drainage and imbibition. In Figure 8.6 the drainage curve corresponds to the primary drainage we discussed in connection with Figure 8.4 in which a non-wetting hydrocarbon phase migrates into a water-wetting porous medium that is completely saturated by water. After the water has been drained to its irreducible saturation, water is injected to flush out the oil. In this particular illustration, k_{rw} exhibits no hysteretic behavior, whereas the imbibition and drainage curves deviate significantly for k_{ro} .

Two-phase relative permeability

It is quite common to use simple analytic relationships to represent relative permeabilities. These are usually stated using the normalized or effective saturation \hat{S}_w from (8.10). The simplest model possible is a pure power-law relationship, which is sometimes called a Corey model,

$$k_{rw} = (\hat{S}_w)^{n_w} k_w^0, \quad k_{ro} = (1 - \hat{S}_w)^{n_o} k_o^0, \quad (8.15)$$

where the exponents $n_w > 1$ and $n_o > 1$ and the constants k_α^0 used for end-point scaling are fitting parameters. Another popular choice is the Brooks–Corey functions

$$k_{rw} = (\hat{S}_w)^{n_1 + n_2 n_3}, \quad k_{ro} = (1 - \hat{S}_w)^{n_1} [1 - (\hat{S}_w)^{n_2}]^{n_3}, \quad (8.16)$$

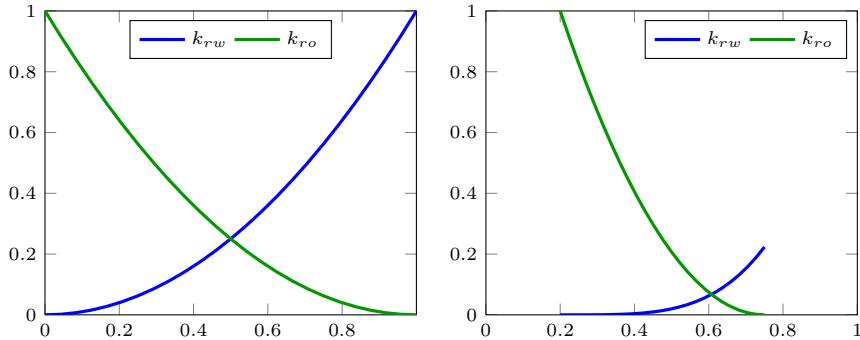


Fig. 8.5. Illustration of relative permeabilities for a two-phase system. The left plot shows an idealized system with no residual saturations, while the right plot shows more realistic curves from a water-wet system.

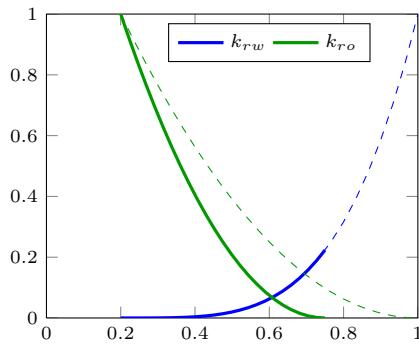


Fig. 8.6. Illustration of relative permeability hysteresis, with drainage curves shown as thin, dashed lines and imbibition curves shown as thick lines.

where, in particular, $n_1 = 2$, $n_2 = 1 + 2/n_b$ and $n_4 = 1$ for the Brooks–Corey–Burdine model and $n_1 = \eta$, $n_2 = 1 + 1/n_b$ and $n_3 = 2$ for the Brooks–Corey–Mualem model. It is also possible to derive models that correspond with the van Genuchten capillary functions in (8.12), which in the case that $m_g = 1 - 1/n_g$ reads

$$\begin{aligned} k_{rw} &= \hat{S}_w^\kappa [1 - (1 - \hat{S}_w^{1/m_g})^{m_g}]^2, \\ k_{ro} &= (1 - \hat{S}_w)^\kappa [1 - \hat{S}_w^{1/m_g}]^{2m_g}, \end{aligned} \quad (8.17)$$

where the connectivity factor κ is a fitting parameter. This is called the van Genuchten–Mualem model, whereas the closed-form expressions obtained for $m_g = 1 - 2/n_g$,

$$\begin{aligned} k_{rw} &= \hat{S}_w^2 [1 - (1 - \hat{S}_w^{1/m_g})^{m_g}], \\ k_{ro} &= (1 - \hat{S}_w)^2 [1 - \hat{S}_w^{1/m_g}]^{m_g}, \end{aligned} \quad (8.18)$$

is called the van Genuchten–Burdine model.

Three-phase relative permeability

Measuring relative permeability has traditionally been costly and involved. Recently, the laboratory techniques have made great progress by using computer tomography and nuclear magnetic resonance (NMR) to scan the test cores where the actual phases are being displaced. However, although standard experimental procedures exist for measuring relative permeabilities in two-phase systems, there is still a significant uncertainty concerning the relevance of the experimental values found and it is difficult to come up with reliable data to be used in a simulator. This is mainly due to boundary effects. Particularly for three-phase systems, there is a lot of uncertainty in reported data (see review by [24],) and it is not unfair to say that no reliable experimental technique exists. Thus, three-phase relative permeabilities are usually modelled using two-phase measurements, for which several theoretical models have been proposed. Most of them are based on an idea first proposed by Stone [236], and seek to combine sets of two-phase relative permeabilities to give three-phase relationships.

In Stone's original model, which was developed for water-wet porous media and is commonly referred to as the Stone I model, the idea was that the relative permeability of water and gas would be identical to the models measured in water/oil displacements and oil/gas displacements, respectively. In the following, I use the convention that k^{wo} denotes a relationship derived from water/oil displacements and k^{og} are relationships derived from oil/gas displacements. Hence, we have that

$$k_{rw}(S_w) = k_{rw}^{wo}(S_w) \quad k_{rg}(S_g) = k_{rg}^{og}(S_g) \quad (8.19)$$

The three-phase relative permeability, on the other hand, depends nonlinearly on both water and gas saturations,

$$k_{ro}(S_w, S_g) = \frac{\hat{S}_o}{k_{ro}^{wo}(S_{wc})} \frac{k_{ro}^{wo}(S_w)}{1 - \hat{S}_w} \frac{k_{ro}^{og}(S_g)}{1 - \hat{S}_g}. \quad (8.20)$$

Here, we have introduced the following scaled quantities

$$\begin{aligned} \hat{S}_o &= \frac{S_o - S_{om}}{1 - S_{wc} - S_{om}}, & \text{for } S_o \geq S_{om} \\ \hat{S}_w &= \frac{S_w - S_{wc}}{1 - S_{wc} - S_{om}}, & \text{for } S_w \geq S_{wc} \\ \hat{S}_g &= \frac{S_g}{1 - S_{wc} - S_{om}} \end{aligned}$$

where S_{wc} is the connate water saturation and S_{om} is the minimum oil saturation, for which several expressions exist. A few years later, Stone proposed an alternative expression for the relative permeability of oil called the Stone II model [237]. A number of modifications and alternative formulations have been developed over the years, see e.g., Baker [24] for a review.

We will come back to the particular three-phase relative-permeability model implemented in MRST in more detail later in Section 11.2 when discussing multiphase compressible flow. This is the same model as the default choice in the commercial Eclipse simulator [227].

8.2 Flow equations for multiphase flow

Having introduced the new physical parameters and key phenomena that characterize multiphase flow of immiscible fluids, we are in a position to develop mathematical models describing multiphase flow. To develop flow models, we will follow more or less the same steps as we did for single-phase flow in Section 4.2. Stating the generic equations that describe multiphase flow is straightforward and will result in a system of partial differential equations that contains more unknowns than equations and hence must be extended with constitutive equations to relate the various physical quantities as well as boundary conditions and source terms (see Sections 4.3.1 and 4.3.2) that describe external forces driving the flow. However, as indicated in the introduction of the chapter, the generic flow model has a complex mathematical character and contains delicate balances of various physical forces whose characteristics and individual strengths vary a lot across different flow regimes. To reveal the mathematical character, but also to make models that are computationally tractable in practical simulations, one will need to develop more specific that apply under certain assumptions on the flow regimes.

8.2.1 Single-component phases

To develop a generic system of flow equations for multiphase flow, we use the fundamental principle of mass conservation. For a system of N immiscible fluid phases that each consists of a single component, we write one conservation equation for each phase,

$$\frac{\partial}{\partial t} (\phi \rho_\alpha S_\alpha) + \nabla \cdot (\rho_\alpha \vec{v}_\alpha) = \rho_\alpha q_\alpha, \quad (8.21)$$

Here, each phase can contain multiple chemical species, but these can be considered as a single component since there is no transfer between the phases and the composition of each phase remains constant in time.

As for the flow of a single fluid, the primary constitutive relationship used to form a closed model is Darcy's law (4.2), which can be extended to multiphase flow by using the concept of relative permeabilities discussed above

$$\vec{v}_\alpha = -\frac{K k_{r\alpha}}{\mu_\alpha} (\nabla p_\alpha - g \rho_\alpha \nabla z). \quad (8.22)$$

This extension of Darcy's law to multiphase flow is often attributed to Muskat and Wyckoff [185] and has only been rigorously derived from first principles

in the case of two fluid phases. Equation (8.22) must therefore generally be considered as being phenomenological. Darcy's law is sometimes stated with the opposite sign for the gravity term, but herein we use the convention that g is positive constant. Likewise, it is common to introduce the phase mobilities $\lambda_\alpha = Kk_{r\alpha}/\mu_\alpha$ or the relative phase mobilities $\lambda_\alpha = \lambda_\alpha K$ to simplify the notation slightly. From the discussion in the previous section, we also have the additional closure relationship stating that the saturations sum to zero (8.1) as well as relations of the form (8.8) that relate the pressures of the different phases by specifying the capillary pressures as functions of the fluid saturations.

Most commercial reservoir simulators compute approximate solutions by inserting the multiphase Darcy equations (8.22) into (8.21), introducing functional relationships for capillary pressure and how ρ_α and ϕ depend on phase pressure, and then discretizing the resulting conservation equations more or less directly. If we use the discrete derivative operators from Section 4.4.2 combined with a backward discretization of the temporal derivatives, the resulting system of discrete equations for phase α reads

$$\frac{(\phi S_\alpha \rho_\alpha)^{n+1} - (\phi S_\alpha \rho_\alpha)^n}{\Delta t^n} + \operatorname{div}(\rho v)_\alpha^{n+1} = (\rho q)_\alpha^{n+1}, \quad (8.23a)$$

$$v_\alpha^{n+1} = -\frac{K k_{r\alpha}}{\mu_\alpha^{n+1}} [\operatorname{grad}(p_\alpha^{n+1}) - g \rho_\alpha^{n+1} \operatorname{grad}(z)], \quad (8.23b)$$

Here, $\phi, S_\alpha, p_\alpha \in \mathbb{R}^{n_c}$ denote the vector with one porosity value, one saturation, and one pressure value per cell, respectively, while v_α is the vector of fluxes of phase α per face, and so on. For properties that depend on pressure and saturation we have for simplicity not introduced any notation to distinguish whether these are evaluated in cells or at cell interfaces. The superscript refers to discrete times at which one wishes to compute the unknown reservoir states and Δt denotes the distance between two such consecutive points in time. The main advantage of using this direct discretization is that it will generally give a reasonable approximation to the true solution as long as we are able to solve the resulting nonlinear system at each time step. To do this, we must pick two primary unknowns and perform some kind of linearization; we will come back to more details on this linearization in later chapters. Here, we simply observe that there are many ways to perform the linearization: we can choose the phase saturations as primary unknowns, the phase pressures, the capillary pressures, or some combinations thereof. How difficult it will be to solve for these unknowns will obviously depend on the coupling between the two equations and nonlinearity within each equation. The main disadvantages of discretizing (8.21) directly are that the general system conceals its mathematical nature and that the resulting equations are not well-posed in the case of an incompressible system. In Section 8.3, we will therefore go back to the continuous equations and analyze the mathematical nature of the resulting system for various choices of primary variables and simplifying assumptions.

8.2.2 Multicomponent phases

In many cases, each phase may consist of more than one chemical species that are mixed at the molecular level and generally share the same velocity (and temperature). This type of flow differs from the immiscible case since dispersion and Brownian motion will cause the components to redistribute if there are macroscale gradients in the mass fractions. The simplest way to model this is through a linear Fickian diffusion,

$$\vec{J}_\alpha^\ell = -\rho_\alpha S_\alpha D_\alpha^\ell \nabla c_\alpha^\ell, \quad (8.24)$$

where c_α^ℓ is the mass fraction of component ℓ in phase α , ρ_α is the density of phase α , S_α is the saturation of phase α , and D_α^ℓ is the diffusion tensor for component ℓ in phase α . Likewise, the chemical species may interact and undergo chemical reactions, but a description of this is beyond the scope of this book.

For multicomponent, multiphase flow, we are, in principle, free to choose whether we state the conservation of mass for components or for fluid phases. However, if we choose the latter, we will have to include source terms in our balance equations that account for the transfer of components between the phases. This can be a complex undertaking and the standard approach is therefore to develop balance equations for each component. For a system of N fluid phases and M chemical species, the mass conservation for component $\ell = 1, \dots, M$ reads

$$\frac{\partial}{\partial t} \left(\phi \sum_\alpha c_\alpha^\ell \rho_\alpha S_\alpha \right) + \nabla \cdot \left(\sum_\alpha c_\alpha^\ell \rho_\alpha \vec{v}_\alpha + \vec{J}_\alpha^\ell \right) = \sum_\alpha c_\alpha^\ell \rho_\alpha q_\alpha, \quad (8.25)$$

where \vec{v}_α is the superficial phase velocity and q_α is the source term. The system is closed in the same way as for single-component phases, except that we now also have to use that the mass fractions sum to zero, (8.3).

The generic equation system (8.22)–(8.25) introduced above can be used to describe miscible displacements in which the composition of the fluid phases changes when the porous medium undergoes pressure and saturation changes and one has to account for all, or a large majority, of the chemical species that are present in the flow system. For systems that are immiscible or partially miscible, it is common to introduce simplifications by lumping multiple species into pseudo-components or even disregard that a fluid phase may be composed of different chemical species as we will see next.

8.2.3 Black-oil models

In the petroleum industry, the most common approach to simulate oil and gas recovery is to use the so-called black-oil equations, which is a special case of multicomponent, multiphase models with no diffusion among the components. The name refers to the assumption that the various chemical species

can be lumped together to form two pseudo-components at surface conditions, a heavy hydrocarbon component called “oil” and a light hydrocarbon component called “gas”. At reservoir conditions, the two components can be partially or completely dissolved in each other depending on pressure and temperature, forming either one or two phases, a liquid oleic phase and a gaseous phase. In addition to the two hydrocarbon phases, the framework includes an aqueous phase that in the simplest models of this class is assumed to consist of only water. In more comprehensive models, the hydrocarbon components are also allowed to dissolve in the aqueous phase and the water component may be dissolved or vaporized in the two hydrocarbon phases. The hydrocarbon fluid composition, however, remains constant for all times.

We will henceforth assume three phases and three components (oleic, gaseous, and aqueous). By looking at our model (8.22)–(8.25), we see that we so far have introduced twenty-seven unknown physical quantities: nine mass fractions c_α^ℓ and three of each of the following quantities ρ_α , S_α , \vec{v}_α , p_α , μ_α , and $k_{r\alpha}$. In addition, the porosity will typically depend on pressure as discussed in Section 2.4.1. To determine these twenty-seven unknowns, we have the following equations: three continuity equations (8.25), an algebraic relation for the saturations (8.1), three algebraic relations for the mass fractions (8.3), and Darcy’s law (8.22) for each of the three phases. Altogether, this constitutes only ten equations. Thus, we need to add seventeen extra closure relations to make a complete model.

The first five of these can be obtained immediately from our discussion in Sections 8.1.3 and 8.1.4, where we saw that the relative permeabilities $k_{r\alpha}$ are functions of the phase saturations and that capillary pressures are functions of saturation and can be used to relate phase pressures as follows

$$p_o - p_w = P_{cow}(S_w, S_o), \quad p_g - p_o = P_{cgo}(S_o, S_g).$$

These required functional forms are normally obtained from a combination of physical experiments, small-scale numerical simulations, and analytical modelling based on bundle-of-tubes arguments, etc. The viscosities are either constant or can be established as pressure-dependent functions through laboratory experiments, which gives us another three equations. The remaining nine closure relations are obtained from mixture rules and PVT models that are generalizations of the equations-of-state discussed in Section 4.2.

By convention, the black-oil equations are formulated as mass conservation for the oil, water, and gas components. To this end, we will employ a simple PVT model that uses pressure-dependent functions to relate fluid volumes at reservoir and surface conditions. Specifically, we use the so-called *formation-volume factors* $B_\ell = V_\ell/V_{\ell s}$ that relate the volumes V_ℓ and $V_{\ell s}$ occupied by a bulk of component ℓ at reservoir and surface conditions, respectively. The formation-volume factors, and their reciprocal *shrinkage factors* $b_\ell = V_{\ell s}/V_\ell$ which some prefer to use for notational convenience, are assumed to depend on phase pressure. In dead-oil systems, the oil is at such a low pressure that it does not contain any gas or has lost its volatile components (which presumably have

escaped from the reservoir). Neither of the components are therefore dissolved in the other phases at reservoir conditions. In so-called live-oil systems, gas is dissolved in the oleic phase. When underground, hydrocarbon molecules is mostly in liquid phase, but gaseous components are liberated when the oil is pumped to the surface. The solubility of gas in oil is usually modelled through the pressure-dependent solution gas-oil ratio, $R_s = V_{gs}/V_{os}$ defined as the volume of gas, measured at standard conditions, that at reservoir conditions is dissolved in a unit of stock-tank oil. In condensate reservoirs, oil is vaporized in the gaseous phase, so that when underground, condensate oil is mostly a gas, but condenses into a liquid when pumped to the surface. The solubility of oil in gas is modeled as the pressure-dependent vaporized oil-gas ratio R_v defined as the amount of surface condensate that can be vaporized in a surface gas at reservoir conditions. Using this notation¹, the black-oil equations for a live-oil system reads,

$$\begin{aligned}\partial_t [\phi(b_o S_o + b_g R_v S_g)] + \nabla \cdot (b_o \vec{v}_o + b_g R_v \vec{v}_g) - (b_o q_o + b_g R_v q_g) &= 0, \\ \partial_t (\phi b_w S_w) + \nabla \cdot (b_w \vec{v}_w) - b_w q_w &= 0, \\ \partial_t [\phi(b_g S_g + b_o R_s S_o)] + \nabla \cdot (b_g \vec{v}_g + b_o R_s \vec{v}_o) - (b_g q_g + b_o R_s q_o) &= 0.\end{aligned}$$

To model enhanced oil recovery, this system can be extended with additional components representing chemical or microbial species that are added to the injected fluids to mobilize immobile oil by altering wettability and/or to improve sweep efficiency and hence the overall displacement of mobile oil. One may also model species dissolved in the resident fluids. Likewise, the black-oil may be expanded by an additional solid phase to account for salts and other minerals that precipitate during hydrocarbon recovery, and possibly also extended to include an energy equation that accounts for temperature effects.

We will return to a more detailed discussion of the general case of three-phase flow with components that may transfer between phases later in Chapter 11. In the rest of the chapter, we continue to study the special case of two fluid phases with no mass transfer between the phases.

8.3 Model reformulations for immiscible two-phase flow

In this section, we will look at various choices of primary variable for the special case of two immiscible fluids, for which the general system of flow equations (8.21) simplifies to

$$\begin{aligned}\frac{\partial}{\partial t}(\phi S_w \rho_w) + \nabla \cdot (\rho_w \vec{v}_w) &= \rho_w q_w, \\ \frac{\partial}{\partial t}(\phi S_n \rho_n) + \nabla \cdot (\rho_n \vec{v}_n) &= \rho_n q_n.\end{aligned}\tag{8.26}$$

and discuss how the various choices affect the mathematical structure of the resulting coupled system of nonlinear differential equations.

¹ The convention in the petroleum literature is to use B_α rather than b_α .

8.3.1 Pressure formulation

If we choose the phase pressures p_n and p_w as the primary unknowns, we need to express the saturations S_n and S_w as functions of pressure. To this end, we assume that the capillary pressure has a unique inverse function $\hat{S}_w = P_c^{-1}(p_c)$ (see (8.8)) so that we can write

$$S_w = \hat{S}_w(p_n - p_w), \quad S_n = 1 - \hat{S}_w(p_n - p_w). \quad (8.27)$$

Then, we can reformulate (8.26) as

$$\begin{aligned} \frac{\partial}{\partial t}(\phi\rho_w\hat{S}_w) + \nabla \cdot \left(\frac{\rho_w K k_{rw}(\hat{S}_w)}{\mu_w} (\nabla p_w - \rho_w g \nabla z) \right) &= \rho_w q_w, \\ \frac{\partial}{\partial t}(\phi\rho_n(1 - \hat{S}_w)) + \nabla \cdot \left(\frac{\rho_n K k_{rn}(\hat{S}_w)}{\mu_n} (\nabla p_n - \rho_n g \nabla z) \right) &= \rho_n q_n, \end{aligned} \quad (8.28)$$

This system is unfortunately highly coupled and strongly nonlinear. The strong coupling comes from the fact that the difference in our primary variables $p_n - p_w$ enters the computation of \hat{S}_w in the accumulation terms and also in the composite functions $k_{rw}(\hat{S}_w(\cdot))$ and $k_{rn}(1 - \hat{S}_w(\cdot))$ used to evaluate the relative permeabilities. As an example of the resulting nonlinearity, we can look at the van Genuchten model for capillary and relative permeabilities, (8.12) and (8.17). Figure 8.7 shows the capillary pressure and relative permeabilities as function of S as well as the inverse of the capillary pressure and relative permeabilities as function of $p_n - p_w$. Whereas the accumulation function is nonlinear, this nonlinearity is further accentuated when used inside the nonlinear relative permeability functions. The pressure formulation was used in the simultaneous solution scheme originally proposed by Douglas, Peaceman, and Rachford [73] in 1959, but has later been superseded by other formulations that reduce the degree of coupling and improve the nonlinear nature of the equation.

8.3.2 Fractional flow formulation in phase pressure

The strong coupling and much of the nonlinearity seen in the previous formulation can be eliminated if we instead express the system in terms of one phase pressure and one phase saturation. A common choice is to use p_n and S_w , which gives the following system

$$\begin{aligned} \frac{\partial}{\partial t}(\phi S_w \rho_w) + \nabla \cdot \left(\frac{\rho_w K k_{rw}}{\mu_w} (\nabla p_n - \nabla P_c(S_w) - \rho_w g \nabla z) \right) &= \rho_w q_w, \\ \frac{\partial}{\partial t}(\phi(1 - S_w)\rho_n) + \nabla \left(\frac{\rho_n K k_{rn}}{\mu_n} (\nabla p_n - \rho_n g \nabla z) \right) &= \rho_n q_n. \end{aligned} \quad (8.29)$$

To further develop this system of equations, it is common to expand the derivatives to introduce rock and fluid compressibilities, as discussed for the

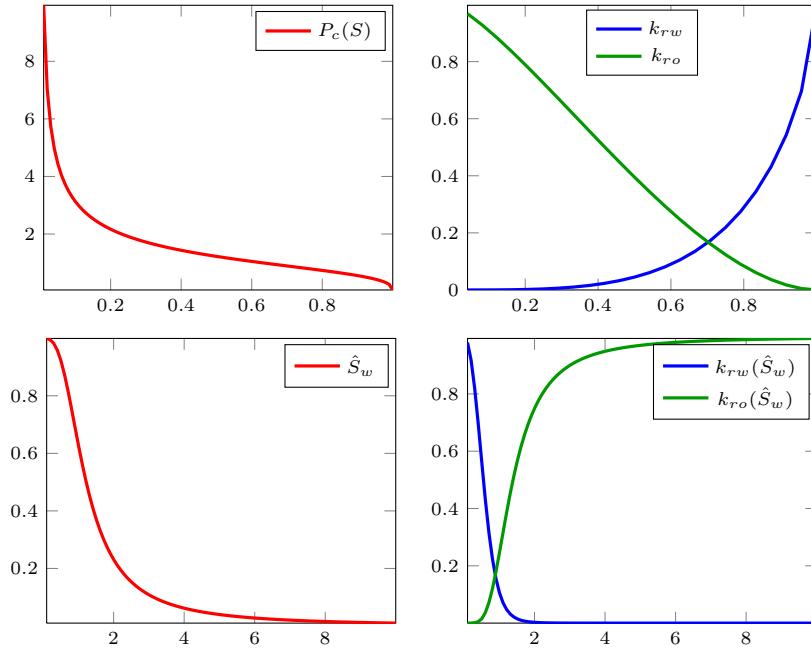


Fig. 8.7. Capillary pressure and relative permeabilities as function of S for the van Genuchten model for $\beta_g = 1$, $m_g = 2/3$, and $\kappa = 1/2$. The lower row shows the saturation and relative permeabilities as function of $p_c = p_n - p_w$.

single-phase flow equation in Section 4.2. First, however, we will look closer at the special case of incompressible flow and develop the so-called *fractional flow* formulation which will enable us to further expose the mathematical structure of the system.

Incompressible flow

For incompressible flow, the porosity ϕ is only a function of time and the fluid densities ρ_α are constant. Using these assumptions, we can simplify the mass-balance equations to be on the form

$$\phi \frac{\partial S_\alpha}{\partial t} + \nabla \cdot \vec{v}_\alpha = q_\alpha. \quad (8.30)$$

To derive the fractional flow formulation, we start by introducing the total Darcy velocity, which we can express in terms of the pressure for the non-wetting phase,

$$\begin{aligned} \vec{v} &= \vec{v}_n + \vec{v}_w = -\boldsymbol{\lambda}_n \nabla p_n - \boldsymbol{\lambda}_w \nabla p_w + (\boldsymbol{\lambda}_n \rho_n + \boldsymbol{\lambda}_w \rho_w) g \nabla z \\ &= -(\boldsymbol{\lambda}_n + \boldsymbol{\lambda}_w) \nabla p + \boldsymbol{\lambda}_w \nabla p_c + (\boldsymbol{\lambda}_n \rho_n + \boldsymbol{\lambda}_w \rho_w) g \nabla z \end{aligned} \quad (8.31)$$

and then add the two continuity equations and use the fact that $S_n + S_w = 1$ to derive a pressure equation

$$\phi \frac{\partial}{\partial t} (S_n + S_w) + \nabla \cdot (\vec{v}_n + \vec{v}_w) = \nabla \cdot \vec{v} = q_n + q_w \quad (8.32)$$

If we now define the total mobility $\lambda = \lambda_n + \lambda_w = \lambda K$ and total source $q = q_n + q_w$, insert (8.31) into (8.32), and collect all terms that depend on pressure on the left-hand side and all other terms on the right-hand side, we obtain an elliptic Poisson-type equation

$$-\nabla \cdot (\lambda K \nabla p_n) = q - \nabla \cdot [\lambda_w \nabla p_c + (\lambda_n \rho_n + \lambda_w \rho_w) g \nabla z], \quad (8.33)$$

which is essentially on the same form as the equation (4.10) that governs single-phase, incompressible flow, except that both the variable coefficient and the right-hand side now depend on saturation through the relative mobility and capillary pressure functions.

To derive an equation for S_w , we multiply each phase velocity by the relative mobility of the other phase, subtract the result, and use Darcy's law to obtain

$$\begin{aligned} \lambda_n \vec{v}_w - \lambda_w \vec{v}_n &= \lambda_w \vec{v}_w - \lambda_n \vec{v}_n \\ &= -\lambda_n \lambda_w K (\nabla p_w - \rho_w g \nabla z) + \lambda_w \lambda_n K (\nabla p_n - \rho_n g \nabla z) \\ &= \lambda_w \lambda_n K [\nabla p_c + (\rho_w - \rho_n) g \nabla z]. \end{aligned}$$

If we now solve for \vec{v}_w and insert into (8.30) for the wetting phase, we obtain what is commonly referred to as the *saturation equation* (or transport equation)

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot [f_w (\vec{v} + \lambda_n \Delta \rho g \nabla z)] = q_w - \nabla \cdot (f_w \lambda_n P'_c \nabla S_w) \quad (8.34)$$

where $\Delta \rho = \rho_w - \rho_n$ and we have introduced the *fractional flow* function $f_w = \lambda_w / (\lambda_n + \lambda_w)$, which measures the fraction of the total flow that consists of the wetting fluid. This is a parabolic equation that accounts for the balance of three different forces, viscous advection $f_w \vec{v}$, gravity segregation $(f_w \lambda_n)(S_w)$, and capillary forces $f_w \lambda_n P'_c \nabla S_w$. The first two terms both involve a first-order derivative and hence have a *hyperbolic* character, whereas the capillary term contains a second-order derivative and hence has a *parabolic* character. We also remark that the negative sign in front of the capillary term is misleading; since $P'_c < 0$ the overall term is positive and hence the parabolic term gives a well-posed problem. Moreover, since λ_n is zero at the end point, the capillary term disappears at this point and hence the parabolic equation degenerates to being hyperbolic. This, in turn, implies that a propagating displacement front in which a wetting fluid displaces a non-wetting fluid, will have finite speed of propagation, and not the infinite speed of propagation that is typical for parabolic problems. Figure 8.8 shows the functional form of

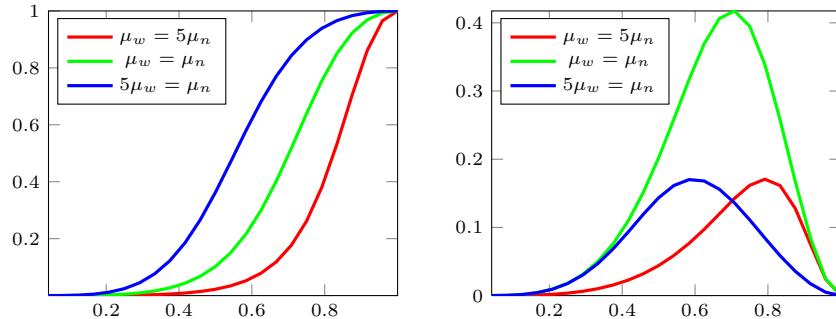


Fig. 8.8. Fractional flow function f_w (left) and gravity segregation function $\lambda_n f_w$ for the van Genuchten–Mualem model (8.17) with $m_g = 2/3$, $\kappa = 1/2$, and $\Delta\rho > 0$.

the two hyperbolic terms for the van Genuchten–Mualem model (8.17). The fractional flow function f_w is monotonically increasing and has a characteristic S-shape. If the flow is dominated by viscous terms so that gravity and capillary forces are negligible, the flow will always be co-current because the derivative of f_w is positive. The gravity segregation function, on the other hand, has a characteristic bell-shape and describes the upward movement of the light phase and downward movement of the heavier phase. (As a consequence, the function is turned upside down if $\rho_w < \rho_n$.) If the gravity term is sufficiently strong, it may introduce counter-current flow. On a field scale, capillary forces are typically small and hence the overall transport equation will have a strong hyperbolic character, and neglecting the capillary term is generally a good approximation. We will return to this model later and discuss a few examples in the special case of purely hyperbolic 1D flow.

The coupling between the elliptic pressure equation and the parabolic saturation equation is much weaker than the coupling between the two continuity equations in the two-pressure formulation (8.28). In the pressure equation (8.33), the coupling to saturation appears explicitly in the effective mobility that makes up the variable coefficient in the Poisson problem and on the right-hand side through the phase mobilities and the derivative of the capillary function. In (8.34), on the other hand, the saturation is only indirectly coupled to the pressure through the total Darcy velocity. In typical displacement scenarios, the saturation variations will be relatively smooth in most of the domain except at the interface between the invading and the displaced fluids and hence the pressure distribution is only mildly affected the evolving fluid saturations. In fact, the pressure and transport equations are completely decoupled in the special case of linear relative permeabilities, no capillary pressure, and equal fluid densities and viscosities. These weak couplings are exploited in many efficient numerical methods that use various forms of operator splitting to solve pressure and fluid transport in separate steps. This is also our method of choice for incompressible flow in MRST; more about this later.

The reformulation discussed above can easily be extended to the case with N immiscible phases, and will result in one pressure equation and a (coupled) system of $N - 1$ saturation equations. Fractional flow formulations can also be developed for multicomponent flow without mass exchange between the phases.

Compressible flow

To develop a fractional flow formulation in the compressible case, we start by expanding the accumulation term in (8.21) and then use fluid compressibilities to rewrite the derivative of the phase densities

$$\begin{aligned}\frac{\partial}{\partial t}(\phi\rho_\alpha S_\alpha) &= \rho_\alpha S_\alpha \frac{\partial\phi}{\partial t} + \phi S_\alpha \frac{\partial\rho_\alpha}{\partial t} + \phi\rho_\alpha \frac{\partial S_\alpha}{\partial t} \\ &= \rho_\alpha S_\alpha \frac{\partial\phi}{\partial t} + \phi S_\alpha \frac{d\rho_\alpha}{dp_\alpha} \frac{\partial p_\alpha}{\partial t} + \phi\rho_\alpha \frac{\partial S_\alpha}{\partial t} \\ &= \rho_\alpha S_\alpha \frac{\partial\phi}{\partial t} + \phi\rho_\alpha S_\alpha c_\alpha \frac{\partial p_\alpha}{\partial t} + \phi\rho_\alpha \frac{\partial S_\alpha}{\partial t}.\end{aligned}$$

If we now insert this into (8.21), divide each equation by ρ_α , and sum the equations, we obtain

$$\frac{\partial\phi}{\partial t} + \phi c_n S_n \frac{\partial p_n}{\partial t} + \phi c_w S_w \frac{\partial p_w}{\partial t} + \frac{1}{\rho_n} \nabla \cdot (\rho_n \vec{v}_n) + \frac{1}{\rho_w} \nabla \cdot (\rho_w \vec{v}_w) = q_t, \quad (8.35)$$

where $q_t = q_n + q_w$. To explore the character of this equation, let us for the moment assume that capillary forces are negligible so that $p_n = p_w = p$ and that the spatial density variations are so small that we can set $\nabla\rho_\alpha = 0$. If we now introduce rock compressibility $c_r = d\ln(\phi)/dp$, (8.35) simplifies to

$$\phi c \frac{\partial p}{\partial t} - \nabla \cdot (\lambda K \nabla p) = \hat{q}. \quad (8.36)$$

Here, we have introduced the total compressibility $c = (c_r + c_n S_n + c_w S_w)$ and a source function \hat{q} that accounts both for volumetric source terms and pressure variations with depth. Equation (8.36) is clearly parabolic, but simplifies to the elliptic Poisson equation (8.33) for incompressible flow ($c = 0$). Fluid compressibilities tend to decrease with increasing pressure, and we should therefore expect that the pressure equation has a strong elliptic character at conditions usually found in conventional reservoirs during secondary and tertiary production, in particular for weakly compressible systems. On the other hand, the pressure evolution will usually have a more pronounced parabolic character during primary depletion and in unconventional reservoirs, where K may be in the range of one micro Darcy or less.

These observations remain true also when capillary forces and spatial density variations are included. However, including these effects makes the mathematical structure of the equation more complicated. In particular, the pressure

equation becomes a *nonlinear* parabolic equation if the spatial variations in density cannot be neglected. To see this, let us once again neglect capillary forces and consider the second last term on the left-hand side of (8.35),

$$\begin{aligned}\frac{1}{\rho_n} \nabla \cdot (\rho_n \vec{v}) &= \nabla \cdot \vec{v}_n + \vec{v}_n \cdot \frac{1}{\rho_n} \nabla \rho_n \\ &= \nabla \cdot \vec{v}_n + \vec{v}_n \cdot (c_n \nabla p_n)\end{aligned}$$

By using Darcy's law, we can transform the inner product between the phase flux and the pressure gradient to involve a quadratic term in ∇p_n or \vec{v}_n ,

$$\begin{aligned}\vec{v}_n \cdot (c_n \nabla p_n) &= -\boldsymbol{\lambda}_n (\nabla p_n - \rho_n g \nabla z) \cdot (c_n \nabla p_n) \\ &= c_n \vec{v}_n \cdot (-\boldsymbol{\lambda}_n^{-1} \vec{v}_n + \rho_n g \nabla z).\end{aligned}$$

The last term on the left-hand side of (8.35) has a similar form and hence we will obtain a nonlinear equation if at least one of the fluid phases have significant spatial density variations.

Going back to (8.35), we see that the equation contains both phase pressures and hence cannot be used directly alongside with a saturation equation. There are several ways to formulate a pressure equation that it only involves a single unknown pressure. We can either pick one of the phase pressures as the primary variable, or introduce an average pressure $p_a = (p_n + p_w)/2$ as suggested in [210]. With p_n as the primary unknown, the full pressure equation reads

$$\begin{aligned}\phi c \frac{\partial p_n}{\partial t} - \left[\frac{1}{\rho_n} \nabla \cdot (\rho_n \boldsymbol{\lambda}_n \nabla p_n) + \frac{1}{\rho_w} \nabla \cdot (\rho_w \boldsymbol{\lambda}_w \nabla p_n) \right] \\ = q_n + q_w - \frac{1}{\rho_w} \nabla \cdot (\rho_w \boldsymbol{\lambda}_w \nabla P_c) + c_w S_w \frac{\partial P_c}{\partial t} \\ - \frac{1}{\rho_n} \nabla \cdot (\rho_n^2 \boldsymbol{\lambda}_n g \nabla z) - \frac{1}{\rho_w} \nabla \cdot (\rho_w^2 \boldsymbol{\lambda}_w g \nabla z).\end{aligned}\quad (8.37)$$

Likewise, with p_a as the primary unknown, we get

$$\begin{aligned}\phi c \frac{\partial p_a}{\partial t} - \left[\frac{1}{\rho_n} \nabla \cdot (\rho_n \boldsymbol{\lambda}_n \nabla p_a) + \frac{1}{\rho_w} \nabla \cdot (\rho_w \boldsymbol{\lambda}_w \nabla p_a) \right] \\ = q_t + \frac{1}{2} \left[\frac{1}{\rho_n} \nabla \cdot (\rho_n \boldsymbol{\lambda}_n) - \frac{1}{\rho_w} \nabla \cdot (\rho_w \boldsymbol{\lambda}_w) \right] \nabla P_c + \frac{1}{2} c_w S_w \frac{\partial P_c}{\partial t} \\ - \frac{1}{2} c_n S_n \frac{\partial P_c}{\partial t} - \frac{1}{\rho_n} \nabla \cdot (\rho_n^2 \boldsymbol{\lambda}_n g \nabla z) - \frac{1}{\rho_w} \nabla \cdot (\rho_w^2 \boldsymbol{\lambda}_w g \nabla z).\end{aligned}\quad (8.38)$$

The same type of pressure equation can be developed for cases with more than two fluid phases. Also in this case we can define a total velocity and use this to obtain $N - 1$ transport equations in fractional flow form.

The saturation equation for compressible flow is obtained exactly in the same way as in the incompressible case,

$$\begin{aligned} \frac{\partial}{\partial t}(\phi\rho_w S_w) + \nabla \cdot [\rho_w f_w (\vec{v} + \boldsymbol{\lambda}_n \Delta \rho g \nabla z)] \\ = \rho_w q_w - \nabla \cdot (\rho_w f_w \boldsymbol{\lambda}_n P'_c \nabla S_w). \end{aligned} \quad (8.39)$$

However, whereas total velocity was the only quantity that coupled the saturation equation to the fluid pressure in the incompressible case, we now have coupling through the porosity and density, which may both depend on pressure. This generally makes the compressible case more challenging than the incompressible case.

8.3.3 Fractional flow formulation in global pressure

The formulation in phase pressure discussed above has a relatively strong coupling between pressure and saturation in the presence of capillary forces. The strong coupling is mainly caused by ∇p_c term on the right-hand side if (8.33). Let us therefore go back and see if we can eliminate this term by making a different choice of primary variables. To this end, we start by looking at the definition of the total velocity, which couples the pressure to the saturation equation:

$$\begin{aligned} \vec{v} &= -\boldsymbol{\lambda}_n \nabla p_n - \boldsymbol{\lambda}_w \nabla p_w + (\rho_n \boldsymbol{\lambda}_n + \rho_w \boldsymbol{\lambda}_w) g \nabla z \\ &= -(\boldsymbol{\lambda}_n + \boldsymbol{\lambda}_w) \nabla p_n + \boldsymbol{\lambda}_w \nabla(p_n - p_w) + (\rho_n \boldsymbol{\lambda}_n + \rho_w \boldsymbol{\lambda}_w) g \nabla z \\ &= -(\boldsymbol{\lambda}_n + \boldsymbol{\lambda}_w) (\nabla p_n - f_w \nabla p_c) + (\rho_n \boldsymbol{\lambda}_n + \rho_w \boldsymbol{\lambda}_w) g \nabla z. \end{aligned}$$

If we now introduce a new pressure variable p , called the *global pressure* [53], defined so that $\nabla p = \nabla p_n - f_w \nabla p_c$, we see that the total velocity can be related to the global pressure through an equation that looks like Darcy's law

$$\vec{v} = -\boldsymbol{\lambda} (\nabla p - (\rho_w f_w + \rho_n f_n) g \nabla z). \quad (8.40)$$

By adding the continuity equations (8.30), and using (8.40), we obtain an elliptic Poisson-type equation for the global pressure

$$-\nabla \cdot [\lambda K (\nabla p - (\rho_w f_w + \rho_n f_n) g \nabla z)] = q_t. \quad (8.41)$$

The advantages of this formulation is that it is very simple and highly efficient in the incompressible case, where pressure often is treated as an immaterial property and one is mostly interested in obtaining a velocity field for the saturation equation. The disadvantages are that it is not obvious how one should specify and interpret boundary conditions for the global pressure and that the global pressure values cannot be used directly in well models (of Peaceman type). The global pressure formulation can also be extended to variable densities and to three-phase flow; the interested reader should consult [58, 56] and references therein. The global formulation is often used by academic researchers because of its simplicity, but is, to the best of my knowledge, hardly used for practical simulations in the industry.

8.3.4 Fractional flow formulation in phase potential

The two fractional flow formulations discussed above are generally well suited to study two-phase immiscible flow with small or negligible capillary forces. Here, flow is mainly driven through high-permeable regions and flow paths are to a large extent determined by the permeability distribution, the shape of the relative permeability functions, and density differences that determine the relative importance of gravity segregation. In other words, the flow is mainly governed by viscous forces and gravity segregation, which constitute the hyperbolic part of the transport equation, whereas capillary forces mostly contribute to adjust the width of the interface between the invading and displaced fluids. For highly heterogeneous media with strong contrasts in capillary functions, on the other hand, the capillary forces may have pronounced impact on the flow paths by enhancing cross-flow in stratified media or reducing the efficiency of gravity drainage. Likewise, because capillary pressure is continuous across geological interfaces, differences in capillary functions between adjacent rocks of different type may introduce (strong) discontinuities in the saturation distribution.

An alternative fractional flow formulation was proposed in [111] to study heterogeneous media with strong contrasts in capillary functions. The formulation is similar to the ones discussed above, except that we now work with fluid potentials rather than fluid pressures. The fluid potential for phase α is given by

$$\Phi_\alpha = p_\alpha - \rho_\alpha g z \quad (8.42)$$

whereas the capillary potential is defined as

$$\Phi_c = \Phi_n - \Phi_w = p_c + (\rho_w - \rho_n) g z. \quad (8.43)$$

By manipulating the expression for the total velocity, we can write it as a sum of two new velocities, that are given by gradients of Φ_w and Φ_c :

$$\begin{aligned} \vec{v} &= -\boldsymbol{\lambda}_n(\nabla p_n - \rho_n g \nabla z) - \boldsymbol{\lambda}_w(\nabla p_w - \rho_n g \nabla z) \\ &= -\boldsymbol{\lambda}_n(\nabla p_w + \nabla p_c - (\rho_n - \rho_w + \rho_n)g \nabla z) - \boldsymbol{\lambda}_w(\nabla p_w - \rho_n g \nabla z) \\ &= -(\boldsymbol{\lambda}_n + \boldsymbol{\lambda}_w)(\nabla p_w - \rho_w g \nabla z) - \boldsymbol{\lambda}_n(\nabla p_c + (\rho_w - \rho_n)g \nabla z) \\ &= -\boldsymbol{\lambda} \nabla \Phi_w - \boldsymbol{\lambda}_n \nabla \Phi_c = \vec{v}_a + \vec{v}_c. \end{aligned}$$

Here, we observe that

$$\vec{v}_w = -\boldsymbol{\lambda}_w \nabla \Phi_w = -\frac{\lambda_w}{\lambda} \boldsymbol{\lambda} \nabla \Phi_w = f_w \vec{v}_a$$

which means that the velocity \vec{a} represents the same phase differential as the phase velocity of the wetting phase, except that it is multiplied by the total mobility, which is smoother and less varying function than the mobility λ_w of the water phase.

Proceeding analogously as above, we can derive the following coupled system for two-phase, immiscible flow:

$$-\nabla \cdot (\lambda K \nabla \Phi_w) = q + \nabla \cdot (\lambda_n K \nabla \Phi_c) \quad (8.44)$$

$$\phi \frac{\partial}{\partial t} + \nabla \cdot (f_w \vec{v}_a) = q_w. \quad (8.45)$$

Compared with the other two fractional flow formulations, this system is simpler in the sense that capillary forces are only accounted for in the pressure equation. Equally important, the saturation equation is purely hyperbolic and only contains advective flow along \vec{v}_a . This makes the saturation much simpler to solve, since one does not have to resolve delicate balances involving nonlinear functions for gravity segregation and capillary forces and can thus easily employ simple upwind discretizations and highly efficient solvers that have been developed for purely co-current flow, like streamline methods [69] or methods based on optimal ordering [186, 157]. On the other hand, the advective velocity \vec{v}_a may vary significantly with time and hence represent a strong coupling between the pressure and transport.

8.3.5 Richards' equation

Another special case arises when modeling the vadose or unsaturated zone, which extends from the ground surface to the groundwater table, i.e., from the top of the Earth to the depth at which the hydrostatic pressure of the groundwater equals one atmosphere. Soil and rock in the vadose zone will generally contain both air and water in its pores, and the vadose zone is the main factor that controls the movement of water from the ground surface to aquifers, i.e., the saturated zone beneath the water table. In the vadose zone, water is retained by a combination of adhesion and capillary forces, and in fine-grained soil one can find pores that are fully saturated by water at a pressure less than one atmosphere.

Because of the special conditions in the vadose zone, the general flow equations modeling two immiscible phases can be considerably simplified. First of all, we can expect that any pressure differences in air will be equilibrated almost instantaneously relative to water since air typically is much less viscous than water. (At a temperature of 20 °C, the air viscosity is approximately 55 times smaller than the water viscosity.) Secondly, air will in most cases form a continuous phase so that all parts of the pore space in the vadose zone is connected to the atmosphere. If we neglect variations in the atmospheric pressure, the pore pressure of air can therefore be assumed to be constant. For simplicity, we can set the atmospheric pressure to be zero, so that $p_a = 0$ and $p_c = p_a - p_w = -p_w$. This, in turn, implies that the water saturation (and the water relative permeability) can be defined as functions of the water pressure using one of the models for capillary pressure presented in Section 8.1.3. Inserting all of this into (8.26), we obtain

$$\frac{\partial}{\partial t}(\phi\rho_w S_w(p_w)) + \nabla \cdot [\rho_w \lambda_{rw}(p_w) K(\nabla p_w - \rho_w g \nabla z)] = 0.$$

If we further expand the accumulation term, neglect spatial gradients of the water density, we can divide the above equation by ρ_w to obtain the so-called *generalized Richards'* equation

$$C_{wp}(p_w) \frac{\partial p_w}{\partial t} + \nabla \cdot [\lambda_{rw}(p_w) K \nabla (p_w - \rho g z)] = 0, \quad (8.46)$$

where $C_{wp} = c_w \theta_w + d(\theta_w)/dp_w$ is a storage coefficient and $\theta_w = \phi S_w$ is the water content. In hydrology, it is common to write the equation in terms of the pressure head

$$C_{wh}(h_w) \frac{\partial h_w}{\partial t} + \nabla \cdot [\kappa_w k_{rw}(h_w) \nabla (h_w - z)] = 0, \quad (8.47)$$

where $C_{wh} = \rho_w g C_{wp}$ and κ_w is the hydraulic conductivity of water (see Section 4.1 on page 119). If we further neglect water and rock compressibility, we obtain the pressure form of the classical Richards' equation [223]

$$C_{ch}(h_w) \frac{\partial h_w}{\partial t} + \nabla \cdot [\kappa_w k_{rw}(h_w) \nabla (h_w - z)] = 0, \quad C_{ch} = \frac{d\theta_w}{dh_w}. \quad (8.48)$$

The corresponding equation for the water flux

$$\vec{v}_w = -\kappa_w k_{rw}(h_w) \nabla (h_w - z) \quad (8.49)$$

was suggested earlier by Buckingham [49] and is often called the Darcy–Buckingham equation. Richards' equation can also be expressed in two alternative forms,

$$\begin{aligned} \frac{\partial \theta_w}{\partial t} + \nabla \cdot [\kappa_w k_{rw}(h_w) \nabla (h_w - z)] &= 0, \\ \frac{\partial \theta_w}{\partial t} + \nabla \cdot [D(\theta_w) \nabla \theta_w] &= \kappa_w k_{rw}(\theta_w) \nabla z, \end{aligned}$$

where $D(\theta_w) = \frac{dh_w}{d\theta_w} \kappa_w k_{rw}(\theta_w)$ is the hydraulic diffusivity tensor. The latter, called the water-content form, is generally easier to solve numerically than the other forms, but becomes infinite for fully saturated conditions.

Various forms of Richards' equation are widely used to simulate water flow in the vadose zone. However, one should be aware that this class of models has important limitations. First of all, since the air pressure is assumed to be constant, these models cannot describe air flow, and also assume that the air phase in the whole pore space is fully connected to the atmosphere. This may not be the case if the porous medium contains heterogeneities blocking the air flow, e.g., in the form of layers that are almost impermeable to air. Likewise, the derivation of the equations assumed that the mobility of air is negligible compared with water, which may not be the case if the relative permeability

of air is much smaller than that of water. If this is not the case, one should expect significant discrepancies in simulation results obtained from Richards' equations and the general two-phase flow equations.

We note in passing that the official release of MRST does not yet have any implementation of Richards' equations.

8.4 The Buckley–Leverett theory of 1D displacements

To shed more light into the two-phase immiscible flow systems, we derive analytical solutions in a few simple cases. As part of this, you will also get a glimpse of the (wave) theory of hyperbolic conservation laws. This theory plays a very important role in the mathematical and numerical analysis of multiphase flow models, but is not described in great detail herein for brevity. The reader is instead encouraged to consult one of the many excellent textbooks on the topic, e.g., [66, 108, 150, 245, 246]. For three-phase flow models, one question that has received much theoretic interest is the possible presence of elliptic regions, i.e., regions in state space for which the eigenvalues of the system are complex conjugates rather than (distinct) real values. The interested reader may consult, e.g., [124, 123] and references therein.

8.4.1 Horizontal displacement

As our first example, we consider incompressible displacement in a 1D homogeneous and horizontal medium, $x \in [0, \infty)$, with inflow at $x = 0$. In the absence of capillary forces, the two phase pressures are equal and coincides with the global pressure p . The pressure equation simplifies to

$$v'(x) = q, \quad v(x) = -\lambda(x)p'(x).$$

If we assume that there are no volumetric source terms, but a constant inflow rate at $x = 0$, it follows by integration that $v(x)$ is constant in the domain $[0, \infty)$. Under these assumptions, the saturation equation (8.34) simplifies to,

$$\frac{\partial S}{\partial t} + \frac{v}{\phi} \frac{\partial f(S)}{\partial x} = 0. \quad (8.50)$$

Let us use this equation to study the propagation of a constant saturation value. To this end, we consider the total differential of $S(x, t)$,

$$0 = dS = \frac{\partial S}{\partial t} dt + \frac{\partial S}{\partial x} dx,$$

which we can use to eliminate $\partial S / \partial t$ from (8.50),

$$-\frac{\partial S}{\partial x} \left(\frac{dx}{dt} \right) \Big|_{dS=0} + \frac{v}{\phi} \frac{df}{dS} \frac{\partial S}{\partial x} = 0.$$

This gives us the following equation for the path of a value of constant saturation

$$\left(\frac{dx}{dt}\right) \Big|_{dS=0} = \frac{df(S)}{dS}. \quad (8.51)$$

If we assume that f is a function of S only, it follows that states of constant saturations will propagate along straight lines, i.e., along paths given by

$$x(t) = x_0(t_0) + \frac{v}{\phi} \frac{df(S)}{dS} (t - t_0).$$

For simplicity, we will henceforth set $v = \phi = 1$ without lack of generality, which is the same as introducing a new time $t^* = t v / \phi$ to rescale the saturation equation.

To make a well-posed problem, we must pose initial and boundary conditions for the saturation equation (8.50) in the form $S(x, 0) = S_{0,x}(x)$ and $S(0, t) = S_{0,t}(t)$. For the special case of a linear fractional flow function $f(S) = S$, which corresponds to a displacement with linear relative permeabilities and equal velocities, the analytical solution is given on closed form as

$$S(x, t) = \begin{cases} S_{0,x}(x - at), & x \geq at, \\ S_{0,t}(at - x), & x < at, \end{cases}$$

which means that the saturation takes the initial data and the boundary data and transport them unchanged along the x -axis with a unit speed (or a speed equal $f'(S)v/\phi$ in the general unscaled case). This way of constructing a solution is called the *method of characteristics*. Notice, that to get a classical solution, the initial and boundary data must be continuous and matching so that $S_{0,x}(0) = S_{0,t}(0)$.

The classical Buckley–Leverett profile, however, is associated with a initial-boundary value problem of the form

$$\frac{\partial S}{\partial t} + \frac{\partial f(S)}{\partial S} = 0, \quad S(x, 0) = S_i, \quad S(0, t) = S_b, \quad (8.52)$$

where S_b generally is different from S_i . Here, we see that all changes in saturation must originate from the jump in values from S_i to S_b at the point $(x, t) = (0, 0)$ and that these changes, which we henceforth will refer to as waves, will propagate along straight lines. In (8.51), we saw that a wave corresponding to a constant S -value will propagate with a speed that is proportional to the derivative of the fractional flow function f . In most cases, $f(S)$ will have a characteristic S-shape with one inflection point, i.e., a point at which $f'(S)$ has an isolated extremum. In essence, this means that saturation values near the extremum should travel faster than saturation values further away on both sides of the extremum. Thus, if we apply the characteristic equation (8.51) naively, we end up with a multivalued solution as illustrated in Figure 8.9, which is clearly not physical. Indeed, if several waves emanate from the same location in space and time, the speeds of these waves have to

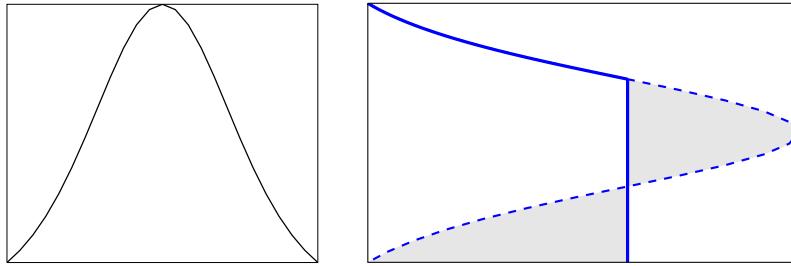


Fig. 8.9. Construction of the self-similar Buckley–Leverett solution $S(x/t)$. The left plot shows $f'(S)$, whereas the right plot shows how the multivalued function given by a naive application of the characteristic equation (8.51) plotted as a dashed line is turned into a discontinuous solution plotted as a solid line by requiring that the shaded areas are of equal size.

be nondecreasing in the direction of flow so that the faster waves move ahead of the slower waves.

The unphysical behavior can be avoided if we simply replace the multivalued solution by a discontinuity, as shown in Figure 8.9. To ensure that mass is conserved, the discontinuity can be constructed graphically so that the two shaded areas are of equal size. This, however, means that the solution cannot be interpreted in the classical sense, since derivatives are not guaranteed to exist pointwise. Instead, the solution must be defined by multiplying the saturation by a smooth function φ of compact support and integrating the result over time and space

$$\iint \left[\frac{\partial S}{\partial t} + \frac{\partial f(S)}{\partial x} \right] \phi(x, t) dx dt = 0.$$

We now integrate by parts and change the order of integration to transfer the derivatives to φ ,

$$\iint \left[S \frac{\partial \varphi}{\partial t} + f(S) \frac{\partial \varphi}{\partial x} \right] dx dt = 0. \quad (8.53)$$

A weak solution is then defined as any solution that satisfies (8.53) for all smooth and compactly supported test functions φ .

In general, propagating discontinuities as shown in Figure 8.9 must satisfy certain conditions. To see this, let $x_d(t)$ denote the path of a propagating discontinuity, and pick two points x_1 and x_2 such that $x_1 < x_d(t) < x_2$. We first use the integral form of the saturation equation to write

$$f(S_1) - f(S_2) = - \int_{x_1}^{x_2} \frac{\partial f(S)}{\partial x} dx = \frac{d}{dt} \int_{x_1}^{x_2} S(x, t) dx \quad (8.54)$$

and then decompose the last integral as follows

$$\int_{x_1}^{x_2} S(x, t) dx = \lim_{\epsilon \rightarrow 0^+} \int_{x_1}^{x_d(t)-\epsilon} S(x, t) dx + \lim_{\epsilon \rightarrow 0^+} \int_{x_d(t)+\epsilon}^{x_2} S(x, t) dx.$$

Since the integrand is continuous in each of the integrals on the right-hand side of the equation, we can use the Leibniz rule to write

$$\frac{d}{dt} \int_{x_1}^{x_d(t)-\epsilon} S(x, t) dx = \int_{x_1}^{x_d(t)-\epsilon} \frac{\partial S}{\partial t} dx + S(x_d(t) - \epsilon, t) \frac{dx_d(t)}{dt}.$$

Here, the first term will vanish in the limit $x_1 \rightarrow x_d(t) - \epsilon$. By collecting terms, substituting back into (8.54), and taking appropriate limits, we obtain what is known as the Rankine–Hugoniot condition

$$\sigma(S^+ - S^-) = f(S^+) - f(S^-), \quad (8.55)$$

where S^\pm denote the saturation values immediately to the left and right of the discontinuity and $\sigma = dx_d/dt$. Equation (8.55) describes a necessary relation between states on opposite sides of a discontinuity, but does not provide sufficient conditions to guarantee that the resulting discontinuity is physically admissible. To this end, one must use a so-called entropy condition to pick out the physically correct solution. If the flux function is strictly convex with $f''(S) > 0$, or strictly concave with $f''(S) < 0$, a sufficient condition is provided by the Lax entropy condition,

$$f'(S^-) > \sigma > f'(S^+), \quad (8.56)$$

which basically states that the discontinuity must be a *self-sharpening* wave in the sense that saturation values in the interval between S^- and S^+ tend to come closer together upon propagation. Such a wave is commonly referred to as a *shock*, based on an analogy from gas dynamics. In the opposite case of a *spreading wave*, commonly referred to as a *rarefaction wave*, nearby states become more distant upon propagation, and for these continuous waves, the characteristic speeds are increasing in the flow direction. In Figure 8.9 this corresponds to the continuous part of the solution that decays towards the shock. For cases with convex or concave flux function, waves arising from discontinuous initial-boundary data will either be shocks or rarefactions, as illustrated in Figure 8.10. If the flux function is linear, or has linear sections, we can also have linearly degenerate waves for which the characteristic speeds are equal on both sides of the wave.

For cases where the flux function is neither strictly convex nor concave, one must use the more general Oleinik entropy condition to single out admissible discontinuities. This condition states that

$$\frac{f(S) - f(S^-)}{S - S^-} > \sigma > \frac{f(S) - f(S^+)}{S - S^+} \quad (8.57)$$

for all values S between S^- and S^+ . For the classical displacement case in which $S_i = 0$ and $S_b = 1$, this condition implies that the correct saturation S^* at the shock front is given by

$$f(S^*)/S^* = f'(S^*), \quad (8.58)$$

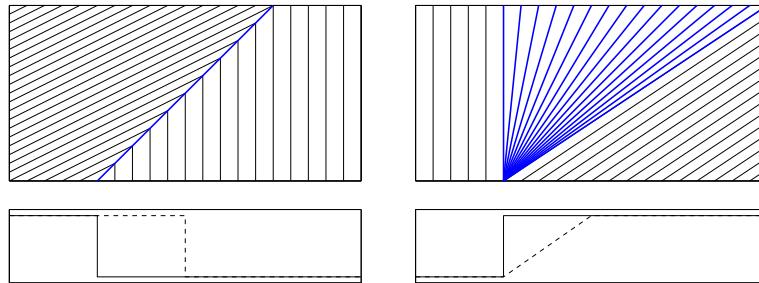


Fig. 8.10. Illustration of self-sharpening (left) and spreading waves (right). The upper plots show the characteristics, with the characteristics belonging to the Riemann fan marked in blue. The lower plots show the initial solution (solid line) and the solution after some time (dashed line).

i.e., the point on the fractional flow curve at which the chord between the points $(0, 0)$ and $(S^*, f(S^*))$ coincides with the tangent at the latter point.

Any reader well acquainted with the theory of hyperbolic conservation laws will probably immediately recognize (8.52) as an example of a Riemann-type problem, i.e., a conservation law with a constant left and right state. Let these states be denoted S_L and S_R . If $S_L > S_R$, the solution of the Riemann problem is a self-similar function given by

$$S(x, t) = \begin{cases} S_L, & x/t < f'_c(S_L), \\ (f'_c)^{-1}(x/t), & f'_c(S_L) < x/t < f'_c(S_R), \\ S_R, & x/t \geq f'_c(S_R), \end{cases} \quad (8.59)$$

which is often referred to as the Riemann fan. Here, $f_c(S)$ is the upper concave envelope of f over the interval $[S_R, S_L]$ and $(f'_c)^{-1}$ the inverse of its derivative. Intuitively, the concave envelope is constructed by imagining a rubber band attached at S_R and S_L and stretched above f in between. When released, the rubber band assumes the shape of the concave envelope and ensures that $f''(S) < 0$ for all $S \in [S_R, S_L]$. The case $S_L < S_R$ is treated symmetrically with f_c denoting the lower convex envelope. Figure 8.10 shows examples of two such Riemann fans, i.e., a single shock in the left plot and a rarefaction fan in the right plot.

Example 8.1. Let us first consider the Buckley–Leverett solution for a pure imbibition, for which the left and right states are $S_L = 1$ and S_R . From the discussion above, we know that the self-similar solution of the associated Riemann problem is found by computing the upper concave envelope of the fraction flux function, which is linear in the interval $[0, S^*]$ and then coincides with $f(S)$ in the interval $[S^*, 1]$, where the saturation S^* behind the shock front is determined from (8.58). If we assume a simple case with relative permeabilities given by the Corey model (8.15) on page 245 with $n_w = n_o = 2$ and $\mu_w/\mu_o = M$, (8.58) reads

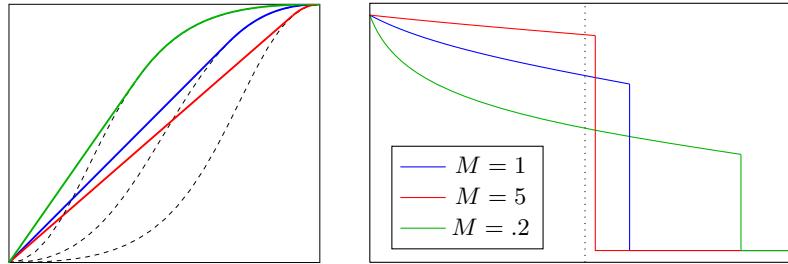


Fig. 8.11. Buckley–Leverett solutions for a pure imbibition case with Corey relative permeabilities with exponent $n_w = n_o = 2$ and mobility ratio $\mu_w/\mu_o = M$. The left plot shows the upper concave envelopes of the flux functions, while the right plot shows the self similar solution $S_w(x/t)$ with $x/t = 1$ shown as a dotted line.

$$\frac{2M(1 - S_M)S_M}{(S_M^2 + M(1 - S_M)^2)^2} = \frac{S_M}{S_M^2 + M(1 - S_M)^2},$$

Here, we have used S_M to denote the front saturation $S^* = \sqrt{M/M+1}$ to signify that it depends on the viscosity ratio M . Figure 8.11 shows the Buckley–Leverett solution for three different viscosity ratio. In all three cases, we see that the solution consists of a shock followed by a trailing continuous rarefaction wave. When the water viscosity is five times higher than oil $M = 5$, the mobility of pure oil is five time higher than the mobility of pure water. Hence, we have a *favorable* displacement in which the water front acts like a piston that displaces almost all the oil at once. In the opposite case with oil viscosity five times higher than the water viscosity, we have an *unfavorable* displacement because the water front is only able to displace a small fraction of the oil. Here, the more mobile oil will tend to finger through the oil and create viscous fingers for displacements in higher spatial dimensions. We will come back to examples that illustrate this later in the book.

In the opposite case of a drainage process, it follows by symmetry that the solution will consist of a shock from 1 to $1 - S_M$ followed by a rarefaction wave from S_M to 0.

If we now consider a case with an injector on the left and a producer at the right, we see that since the saturation is constant ahead of the shock, the amount of produced oil will equal the amount of injected water until the first water reaches the producer. This means that the cumulative oil production has a linear slope until water breakthrough. After water has broken through, the well will be producing a mixture of oil and water and the slope of the cumulative oil decreases.

Let us try to determine the amount of produced oil at water breakthrough. To this end, we can assume that water is injected at $x = 0$ and oil produced at $x = L$. At water breakthrough, the saturation at the producer $S(x = L) = S^*$ and the amount of oil produced equals $L\bar{S}$, where \bar{S} is the average water

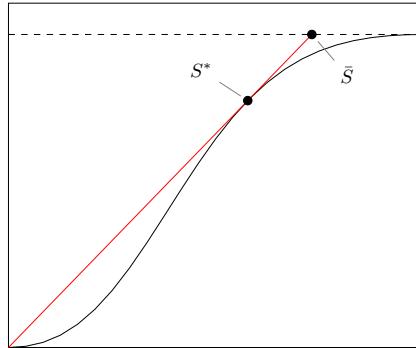


Fig. 8.12. Graphical determination of front saturation and average water saturation at water breakthrough.

saturation defined as

$$\begin{aligned}\bar{S} &= \frac{1}{L} \int_0^L S_w dx = \frac{1}{L} [xS]_{x=0}^{x=L} - \frac{1}{L} \int_1^{S^*} x dS \\ &= S^* - \frac{t}{L} \int_1^{S^*} \left(\frac{dF}{dS} \right) dS = S^* - \frac{t}{L} [f^* - 1]\end{aligned}$$

The second equality follows from the total differential $d(xS) = x dS + S dx$, while the third equality follows from the equation (8.51) that describes the position $x(t)$ of a constant saturation value at time t . Finally, the water breakthrough occurs at time $t = L/f'(S^*)$ and hence we have that,

$$\frac{1 - f(S^*)}{\bar{S} - S^*} = f'(S^*),$$

which can graphically interpreted as follows: S_w is the point at which the straight line used to define the front saturation S^* intersects the line $f = 1$ as shown in Figure 8.12.

8.4.2 Gravity segregation

As our second example of analytical solutions in 1D, we consider a pure gravity displacement, for which the transport equation is on the form

$$\frac{\partial S}{\partial S} + \frac{\partial}{\partial z} \left(\frac{\lambda_n \lambda_w}{\lambda_w + \lambda_n} \right) = 0, \quad (8.60)$$

where we without lack of generality have scaled away the constant $g\Delta\rho$. With a slight abuse of notation, we will henceforth call the flux function for $g(S)$. This flux function will have a characteristic bell-shape, e.g., as shown in Figure 8.13, where the points downward if $\rho_w > \rho_n$ and upward in the opposite case. In the

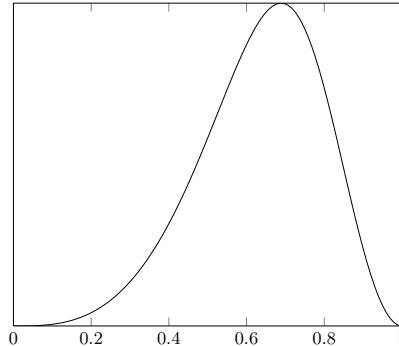


Fig. 8.13. Fractional flow function for 1D gravity segregation for Corey relative permeabilities (8.15) with $n_w = 3$, $n_o = 2$, and mobility ratio $\mu_w = 5\mu_o$ for the case that $\rho_w > \rho_o$.

following, we will assume that the wetting fluid is most dense. Let us consider the flow problem with one fluid placed on top of the other so that the fluids are separated by a sharp interface. In the case that the lightest fluid is on top, we have that $S_L < S_R$ and from the discussion of (8.59) on page 267 we recall that the Riemann solution is found by computing the lower convex envelope of $g(S)$. Here, $g_c(S) \equiv 0$ and hence we have a stable situation. In the opposite case, $S_L > S_R$ and $g_c(S)$ will be a strictly concave function, for which $g'_c(0)' > 0 > g'_c(1)$. This is an unstable situation, where the heavier fluid will start to move downward (in the positive z -direction) and the lighter fluid upward (in the negative z -direction) near the interface. Let us consider the resulting wave patterns in more detail in a specific case:

Example 8.2. We consider a gravity column inside a homogeneous sand body confined by a sealing medium at the top and the bottom. For simplicity, we scale the domain to be $[0, 1]$, assume that the initial fluid interface is at $z = 0.5$ and use a Corey relative permeability model with $n_w = n_o = 2$ and equal mobilities.

This choice of parameters gives a symmetric flux function $g(S)$ as shown in the left plot of Figure 8.14. Our initial condition with a heavy fluid on top of a lighter fluid is described by setting $S_L = 1$ in the top half of the domain and $S_R = 0$ in the bottom half of the domain. As explained above, the Riemann problem is solved by computing the upper concave hull of $g(S)$. The hull $g_c(S)$ is a linear function lying above $g(S)$ between 0 and S_R^* (where S_R^* is given by solution to $g'(S_R^*)S_R^* = g(S_R^*)$), follows the graph of $g(S)$ in the interval S_R^* to S_L^* , and is a linear function lying above $g(S)$ between S_L^* and 1. At the boundaries $x = 0, 1$, we have $g(S_L) = g(S_R) = 0$, and hence no-flow conditions as required. Initially, the solution thus consists of three different regions:

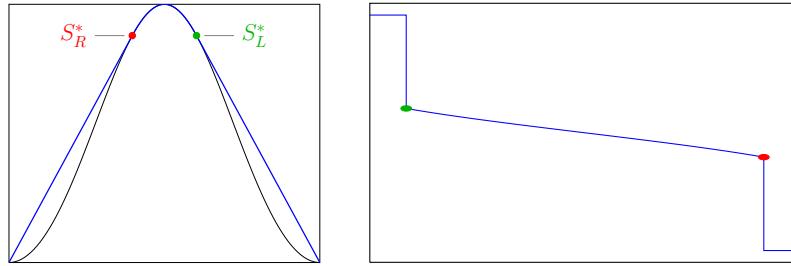


Fig. 8.14. Buckley–Leverett solution for a gravity column with sealing top and bottom before the moving fluids have contacted the top/bottom of the domain. (Model: Corey relative permeabilities (8.15) with $n_w == n_0 = 2$, and mobility ratio $\mu_w = \mu_o$ and $\rho_w > \rho_n$).

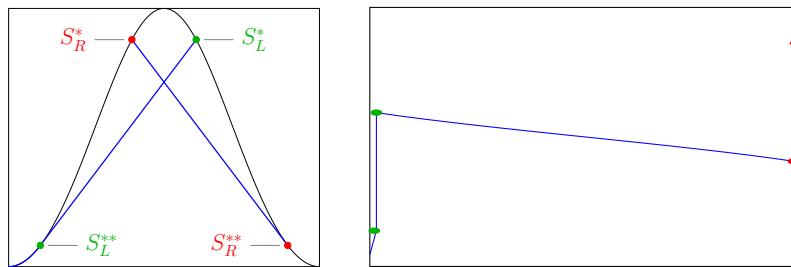


Fig. 8.15. Buckley–Leverett solution for a gravity column with sealing top and bottom just after the light/heavy fluids started to accumulate at the top/bottom of the domain. To ensure no-flow, the solution is $S = 0$ at $z = 0$ and $S = 1$ at $z = 1$.

- a single-phase region on top that only contains the heavier fluid ($S_L = 1$) which is bounded by above by the lines $z = 0$ and below by the position of the shock $1 \rightarrow S_L^*$;
- a two-phase transition region that consists of a mixture of heavier moving downward and lighter fluid moving upward, this corresponds to a centered rarefaction wave in which the saturation decays smoothly from S_L^* to S_R^* ; and
- a single-phase region at the bottom that only contains light fluid. This region is bounded above by a shock $S_R^* \rightarrow 0$ and from below by the line $z = 1$.

As time passes, the two shocks that limit the single-phase regions of heavy and light fluids will move upward and downward, respectively, so that an increasing portion of the fluid column is in a two-phase state, see the right plot in Figure 8.14. Eventually, these shocks will reach the top and the bottom of the domain. Because of the symmetry of the problem, it is sufficient for us to only consider one end of the domain, say $z = 0$. Since we have assumed no flow across the top of the domain, the shock $1 \rightarrow S_L^*$ cannot continue to propagate upward. We thus get a Riemann problem defined by right state S_L^*

and a left state S_L that could by any value that satisfies $g(S_L) = 0$. The left state cannot be 1, since this would give back the same impermissible shock $1 \rightarrow S_L^*$. If the left value is $S_L = 0$ instead, i.e., a single-phase region with only light fluid forming on the top, the Riemann solution will be given by the concave envelope of $g(S)$ over the interval $[0, S_L^*]$ and hence consist fully of waves with positive speed that propagate downward again. The left plot in Figure 8.15 shows the corresponding convex envelope, which we see gives a rarefaction wave 0 to S_L^{**} and a shock $S_L^{**} \rightarrow S_L^*$. As the leading shock propagates downward, it will interact with the upward-moving part of the initial rarefaction wave and form a shock, whose right state gradually decays towards $S = 0.5$. The solution is the bottom of the domain is symmetric, with accumulation of a single-phase region of heavy fluid.

8.4.3 Front tracking: semi-analytical solutions

While it was quite simple to find an analytical solution in closed form for simple Riemann problems like the ones discussed above, it becomes much more complicated to find analytical solutions for general initial data or when different waves start to interact, as we observed on the previous page when the two initial shocks reflected from the top and bottom of the gravity column. Generally, one must therefore resort to numerical discretizations in the form of a finite element, difference, or volume method. However, for one-dimensional saturation equations there is a much more powerful alternative, which we will introduce you to very briefly.

Front tracking, or wave front tracking, is a semi-analytical method in which the key idea is to replace the functions describing initial and boundary values by piecewise constant functions. This way, the solution for a small time interval be given by piecing together solution of local Riemann problems and consist of a set of constant states separated by shocks and rarefaction waves. As we saw above, shocks correspond to the linear parts of the local convex/concave envelopes for each Riemann problem, while rarefaction waves correspond to the nonlinear parts of envelopes. Now, if we further approximate the flux function by a piecewise linear function, we ensure that the convex/concave envelopes are always piecewise linear, so that the solution of Riemann problems will consist only of shocks. In other words, the solution to the continuous, but approximate PDE problem will consist entirely of constant states separated by discontinuities that propagate with a constant speed given by the Rankine-Hugoniot condition (8.55) on page 266. Each time two or more discontinuities ‘collide’, we have a new Riemann problem that will give rise to a new set of discontinuities that emanate out of the collision point. Solving Riemann problems by determining local convex/concave envelopes and keeping track of propagating discontinuities can quite easily be formulated as numerical algorithms, and we end up with a numerical method that is unconditionally stable and formally first order accurate. In fact, one can prove that for any finite spatial domain, there will be a finite number of shock collisions, and

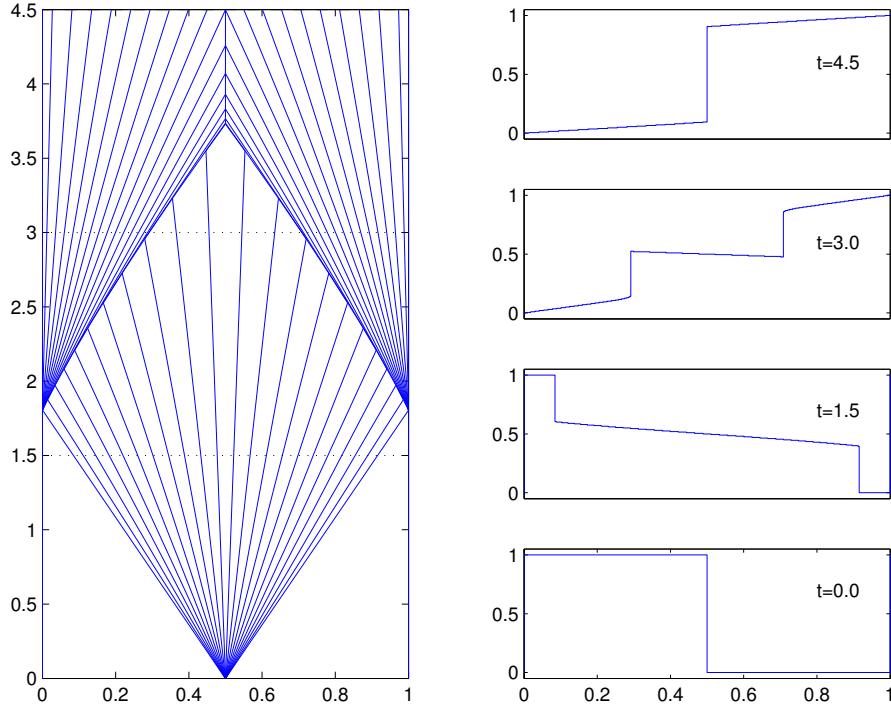


Fig. 8.16. Solution for a gravity column with heavy fluid on top of a light fluid computed by front tracking for a model with Corey relative permeabilities with $n_w = n_o = 2$ and equal viscosities. The left column shows the wave pattern in the (z, t) plane, where each blue line corresponds to the propagation of a discontinuous state. The right column shows the solution $S(z, t)$ at four different times.

hence an approximate solution up to infinite time can therefore be computed in a finite number of steps. See [108] for more details.

In the following, we will use a front-tracking solver developed as part of [126] to study a few simple 1D cases

Example 8.3. We revisit the case studied in Example 8.2: gravity segregation of a heavy fluid placed on top of a lighter fluid inside a sand box with sealing top and bottom. Figure 8.16 shows the approximate solution computed by approximating the flux function with a piecewise linear function with 100 equally spaced segments. Initially, the solution contains two constant states $S_L = 1$ and $S_R = 0$, separated by a composite wave that consists of an upward-moving shock, a centered rarefaction wave, and a downward moving shock, as explained earlier in Example 8.2. Notice that the approximate solution at time $t = 1.5$ is of the same form as shown in Figure 8.14, except that in the approximate front-tracking solution the continuous rarefaction wave is replaced by a sequence of small discontinuities. The right plot in Figure 8.15 illustrated the

form of the exact solution just after the two initial shocks have reflected at the top and bottom of the domain. In Figure 8.16 we show the solution after the reflected waves have propagated some time upward/downward. Here, we clearly see that these waves consist of a fast shock wave followed by a rarefaction wave. As the leading shock interacts with the centered rarefaction wave originating from $(z, t) = (0.5, 0)$, the difference between left and right states will diminish, and the shock speed decay. Later, the two shocks collide and form a stationary discontinuity that will eventually become a stable steady state with the lighter fluid on top and the heavier fluid beneath.

In general, the transport in an inclined reservoir section will be driven by a combination of gravity segregation and viscous forces coming from pressure differentials. In the next example we will study one such case.

Example 8.4 (CO_2 storage). We consider two highly idealized models of CO_2 storage in an inclined aquifer that either has a slow drift of brine in the upslope or in the downslope direction. If the injection point lies deeper than approximately 800 meters below the sea level, CO_2 will appear in the aquifer as a supercritical liquid phase that has much lower density than the resident brine. The injected fluid, which is commonly referred to as the CO_2 plume, will therefore tend to migrate in the upslope direction. As our initial condition, we assume (somewhat unrealistic) that the CO_2 has been injected so that it completely fills a small section of the aquifer $x \in [\frac{1}{4}, \frac{3}{4}]$. To study the migration after injection has ceased, we can now use a fractional flow function of the form,

$$F(S) = \frac{S^2 \mp 4S^2(1-S)^3}{S^2 + \frac{1}{5}(1-S)^3},$$

where the minus sign denotes upslope background drift and the plus sign downslope drift. Figure 8.17 shows the flux functions along with the envelopes corresponding to the two Riemann problems.

Let us first consider the case with upslope drift. At the tip of the plume ($x = \frac{3}{4}$), there will be a drainage process in which the more buoyant CO_2 displaces the resident brine. This gives a composite wave that consists of a shock $S_d^* \rightarrow 1$ that propagates upslope, followed by a centered rarefaction wave from S_d^* to S_d^{**} having wave speeds both upslope and downslope, and a trailing shock wave $0 \rightarrow S_d^{**}$ that propagates in the downslope direction. Likewise, at the trailing edge of the plume, the resident brine will imbibe into the CO_2 giving a wave consisting of a shock $S_i^* \rightarrow 0$ followed by a rarefaction wave, both propagating in the upslope direction. The two Riemann fans can be seen clearly in Figure 8.18, and will stay separated until $t \approx 0.18$, when the shock from the imbibition process collides with the trailing drainage shock. The result of this wave interaction is a new and slightly weaker shock that propagates increasingly faster in the upslope direction, followed by a continuous rarefaction wave, which is an extension of the initial imbibition rarefaction. At time $t \approx 0.665$, the new shock wave has overtaken the tip of

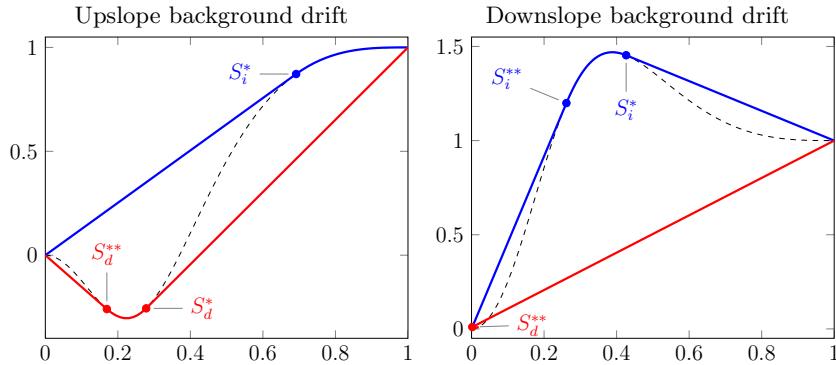


Fig. 8.17. Flux functions for the CO₂ migration example. The dashed curves are the flux function $F(S)$, which accounts for a combination of gravity segregation and viscous background drift. The blue lines show the concave envelopes corresponding to imbibition at the left edge of the plume, while the red lines are the convex envelopes corresponding to drainage at the right edge the plume.

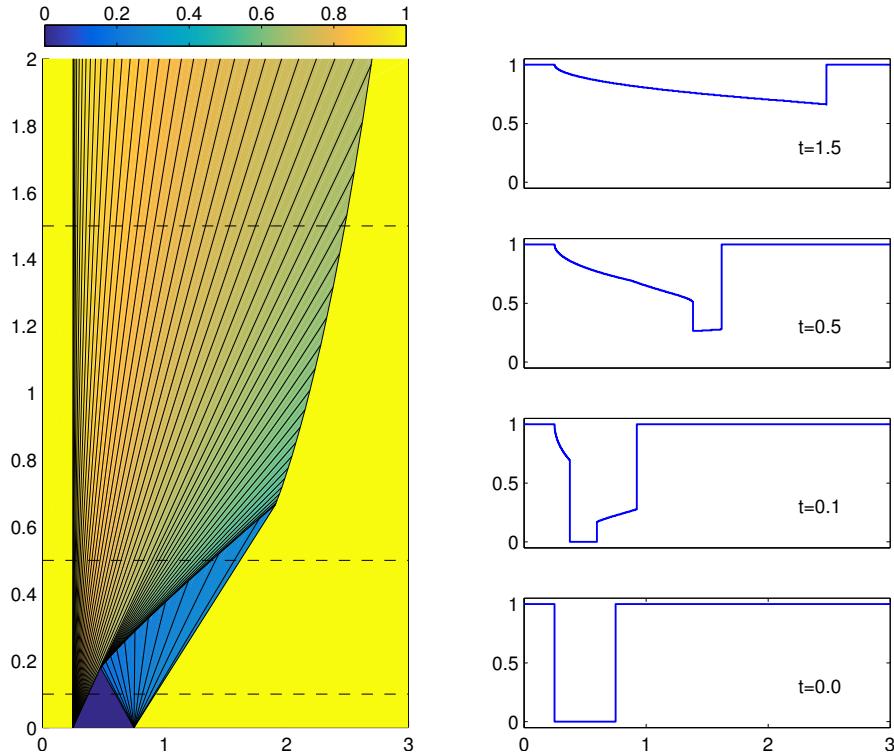


Fig. 8.18. Solution for a conceptual model of CO₂ injected into an inclined 1D aquifer with a background upslope drift. Upslope direction is to the right in the figure.

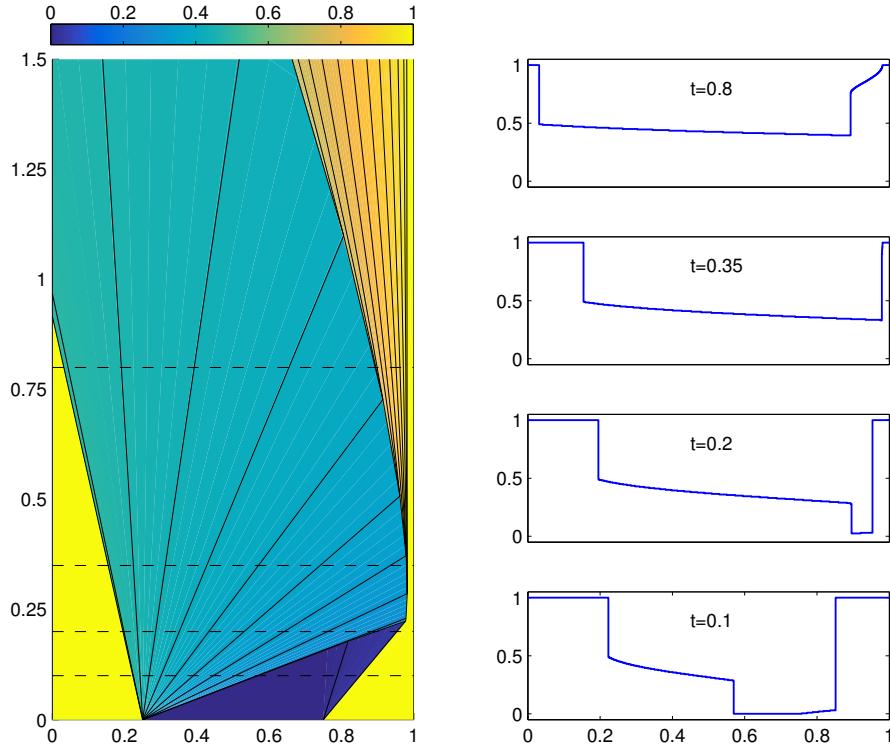


Fig. 8.19. Solution for a conceptual model of CO_2 injected into an inclined 1D aquifer with a background downslope drift. Upslope direction is to the left in the figure.

the plume, and the result of this interaction is that the upslope migration of the plume gradually slows down.

If we remove the upslope drift from the model, the initial imbibition process is no longer present, but the long-term behavior remains the same and consists of a leading shock wave followed by a rarefaction that slowly eats up and slows down the shock.

Figure 8.19 shows the solution in the opposite case, with a downslope background drift. Here, the imbibition process at the left edge of the plume gives a shock $0 \rightarrow S_i^{**}$ that propagates in the downslope direction (to the right in the figure), a centered rarefaction wave from S_i^{**} to S_i^* with waves that propagate both in the downslope and upslope direction, followed by a shock $S_i^* \rightarrow 1$ that propagates upslope. The drainage at the right edge gives a strong shock $S_d^* \rightarrow 1$ that propagates downslope, followed by a weak rarefaction wave from 0 to S_d^* . At time $t \approx .25$ downslope imbibition shock has overtaken the both the rarefaction wave and the shock from the drainage process. The resulting wave interaction first forms a stationary shock with left state S_i^{**} .

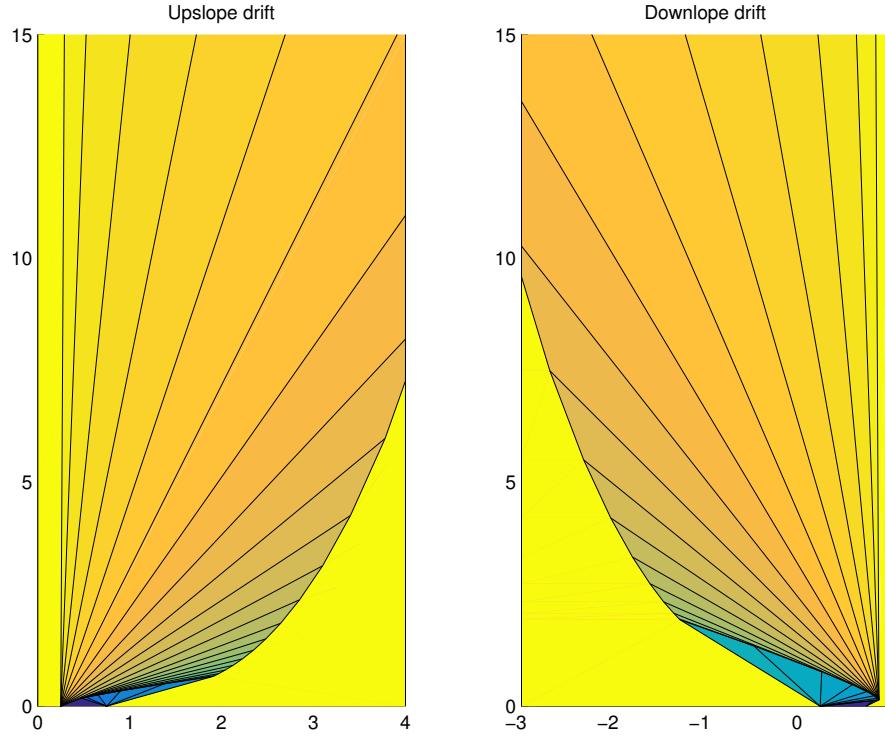


Fig. 8.20. Long-time solutions for the two conceptual models of CO₂ storage.

However, as the positive part of the imbibition rarefaction wave continues to propagate downslope, the left state gradually increases, and at some point the left state passes the value $S = \sqrt{1/10}$ for which $F(S) = 1$. When this happens, the stationary shock turns into a composite wave consisting of a shock and a trailing rarefaction wave that both propagate in the upslope direction.

Figure 8.20 shows the two migration cases in a longer time perspective. We clearly see that injecting CO₂ so that it must migrate upward against a downslope brine drift is advantageous since it will delay the upslope migration of the gradually thinning plume.

The purpose of the previous example was to introduce you to the type of wave patterns that arise in hyperbolic saturation equations, and show that these can be relatively complicated even for simple initial configurations. While front-tracking is probably the best method you can find to study wave patterns in 1D hyperbolic saturation equations, it would not be the method of choice to study multiphase flow models in multiple spatial dimensions. In the next chapter, we will therefore continue to discuss finite-volume methods that are more suitable for multidimensional reservoir simulation. In simulation of real reservoirs, one cannot hope to resolve the dynamics to the level of detail

seen in the last example. To be able to compute shocks and other types of discontinuities, finite-volume (and finite-element) methods contain a certain amount of numerical dissipation that will tend to smear any discontinuity. Likewise, strong heterogeneities and radial flow near wells can introduce orders of magnitude variations in Darcy velocities which will necessitate the use of small time steps or implicit temporal discretizations, which both will tend to smear discontinuities and make it more difficult to resolve complex wave interactions in full detail.

We end our discussion of analytical solutions with a few remarks about the simulation of CO₂ storage, since this was touched upon conceptually in Example 8.4. In more realistic modelling of CO₂ storage, one would need a slightly more advanced model that accounts for the effect that the lighter CO₂ plume will flow to the top of the formation and migrate upslope as a thin layer under the sealing caprock that bounds the aquifer from above. This can be done by either using a 3D or 2D cross-sectional saturation equation, possibly coupled with a pressure equation. However, the most efficient approach to study large-scale, long-term CO₂ migration is to integrate the pertinent flow equations in the vertical direction to form a vertically-averaged model that accounts for the vertical fluid distribution in an averaged sense. MRST has a large module, `co2lab`, that offers a wide variety of vertically-averaged models and other types of reduced models suitable for modeling of large-scale CO₂ storage. A discussion of such models is unfortunately beyond the scope of this book; the interested reader should instead consult one of the papers that describe the methods implemented in `MRST-co2lab` [193, 198, 195, 196, 197, 160, 18].

Discretizing Hyperbolic Transport Equations

Earlier in the book, we have covered a number of discretization and solution techniques. The primary discretization discussed so far, is the standard two-point scheme that is used to approximate second-order elliptic operators of the type $\nabla \cdot \alpha(\vec{x}) \nabla$, which we introduced in Section 4.4.1. However, in Section 4.4.3 we also introduced the upwind scheme for approximating linear advective terms in the stationary time-of-flight and tracer equation. The same operator was later used to approximate the advection of enthalpy in Section 7.5. In Chapter 7.2 we studied time-dependent problems, introduced fully-implicit discretizations for compressible single-phase flow, and discussed how to use a Newton–Raphson method to linearize and solve the resulting system of discrete nonlinear equations. With some modifications, these techniques are all we need to introduce what is the most widespread approach to simulate compressible, multiphase flow. However, if we also want to ensure that our solution procedure is as efficient as possible, we need to take special care when linearizing the equations and in most cases also apply some kind of preconditioner which ensures that our solution algorithm is well-behaved. We will come back to this in more details later in the book.

For incompressible multiphase flow, the situation is slightly different. As we saw in the previous chapter, it is common to write the system in a fractional-flow form that consists of an elliptic pressure equation and one or more parabolic/hyperbolic transport equations for phase saturations and/or chemical components. The same approach can also be applied for compressible flow. The pressure equation then becomes parabolic, but has a certain elliptic character that can be quite strong for compressible flows. One of the main motivations for introducing a fractional-flow formulation to make it more simple to introduce specialized numerical methods for each of the sub-equations. In Chapter 6, you have already encountered a number of more advanced discretization methods that have been developed to handle the elliptic character of pressure equations in a consistent way. For the transport equation, it is common to apply a different set of numerical methods. Extensive research has been devoted over the last 2–3 decades to develop numerical methods

that can handle equations with a strong hyperbolic in an accurate and stable manner. In this chapter we will give you a quick crash course in numerical methods for hyperbolic conservation laws. Since this was the first topic I focused on in my research career, I have indulged myself by also including a bit of theory on hyperbolic equations and their numerical methods. If you are not interested in this wider perspective and only wish to know the simplest possible approach to solving saturation equations, you can jump directly to Section 9.6 on page 306, which discusses the standard upstream mobility-weighting method which has been implemented for general polyhedral grids in the `incomp` module of MRST.

9.1 A new solution concept: entropy-weak solutions

In reservoir simulation, pressure gradients, gravity segregation, capillary forces, source terms from injection and production wells, and boundary conditions all play a prominent role in driving and restricting the transport of fluid phases. In the first part of this chapter, we will nonetheless only consider a simple, homogeneous conservation law in an infinite domain subject to initial data,

$$u_t + f(u)_x = 0, \quad u(x, 0) = u_0(x). \quad (9.1)$$

Here, u is some conserved quantity, which need not necessarily be a fluid saturation, and $f(u)$ is a generic flux function. Equation (9.1) usually arises from a more fundamental physical law on integral form,

$$\frac{d}{dt} \int_{x_1}^{x_2} u(x, t) dx = f(u(x_1, t)) - f(u(x_2, t)), \quad (9.2)$$

which states that the rate of change of quantity u within the interval $[x_1, x_2]$ equals the flux across the ends $x = x_1$ and $x = x_2$ of the interval.

As we have already seen in the previous chapter, solutions to (9.1) may develop discontinuities in finite time, even for smooth initial data if the flux function f is nonlinear. This means that the solution of (9.1) is usually understood in the weak sense,

$$\int_0^\infty \int_{\mathbb{R}} (u\varphi_t + f(u)\varphi_x) dt dx = \int_{\mathbb{R}} u_0(x)\varphi(x, 0) dx. \quad (9.3)$$

Here, $\varphi(x, t)$ is a continuous and smooth test function that has *compact support* so that it vanishes outside a bounded region in the (x, t) -plane.

Solutions defined by the weak form (9.3) are not necessarily unique. The solution concept must therefore be extended to include additional admissibility conditions to single out the correct solution among several possible candidates satisfying the weak form. A classical method to obtain uniqueness is to add a regularizing second-order term to (9.1), giving a parabolic equation

$$u_t^\epsilon + f(u^\epsilon)_x = \epsilon u_{xx}^\epsilon,$$

which has unique, smooth solutions. The unique solution of the hyperbolic equation (9.1) is then defined as the limit of $u^\epsilon(x, t)$ as ϵ tends to zero. In models from fluid dynamics, such a second-order term can be proportional to the viscosity of the fluid, and the method is therefore called the *vanishing viscosity method*. For flow in porous media, the transport equations will already have a second-order term coming from capillary forces (see (8.34) on page 255). However, capillary forces are often neglected when studying flow dominated by global pressure gradients, or the capillary function may vary with rock type and this can introduce discontinuities in the phase saturations at the interface between different rock types.

Since the vanishing viscosity solutions $u^\epsilon(x, t)$ are smooth, it is possible to use techniques from classical analysis to prove the existence, uniqueness, and stability of the solution to (9.1). This was first done in a seminal work by Kružkov [142], which paved the road for the modern theory of nonlinear partial differential equation of hyperbolic–parabolic type. Later in this chapter we will apply the principle of adding a viscous term in a discrete sense to construct stable numerical methods.

Working with limits of viscous solutions is not very practical. Instead, we impose other admissibility conditions. We have already encountered the Lax and Oleinik entropy conditions, (8.56) and (8.57), which were used to single out **permissible discontinuities**. An alternative approach, which concept of entropy functions, which is motivated from thermodynamic conditions in gas dynamics. To this end, we introduce a (convex) entropy function $\eta(u)$ and a corresponding entropy flux $\psi(u)$ and **require that an admissible weak solution u must satisfy the entropy condition**

$$\eta(u)_t + \psi(u)_x \leq 0, \quad (9.4)$$

which must be interpreted in the weak sense as

$$\int_0^\infty \int_{\mathbb{R}} (\eta(u)\varphi_t + \psi(u)\varphi_x) dt dx + \int_{\mathbb{R}} \eta(u_0(x))\varphi(x, 0) dx \geq 0, \quad (9.5)$$

for all nonnegative test functions φ . The functions η and ψ are called an **entropy pair** and satisfy the compatibility condition $\psi'(u) = \eta'(u)f'(u)$. The existence of such entropy pairs is not obvious for a general system of conservation laws, but such pairs exist and have a clear physical interpretation for several important systems of equations, for instance in gas dynamics. For scalar equations it can be shown that all entropy pairs with convex η are equivalent. A common choice is the so-called Kružkov entropy pair, $\eta(u) = |u - k|$ and $\psi = \text{sign}(u - k)(f(u) - f(k))$. The solution $u(x, t)$ is called an *entropy weak solution* of (9.1) if it satisfies (9.5) for all $k \in \mathbb{R}$ and nonnegative test functions φ .

9.2 Conservative finite-volume methods

In Chapter 4, we introduced a finite-volume discretization for the Poisson equation. In this discretization, the unknown functions are represented in terms of average values over a set of finite-volumes. The resulting discrete equations were derived by requiring that the integrated PDE model holds in an averaged sense. This made it simple to formulate a discretization on general polyhedral grids and ensured that the resulting method obeys the important property of mass conservation. We will use a similar finite-volume approach to discretize the conservation law (9.1). This has one additional advantage for hyperbolic conservation laws. As we saw above, the differential equation ceases to be pointwise valid in the classical sense once a discontinuity arises in $u(x, t)$. Hence, we can also expect that the pointwise approximations used in standard finite-difference methods will break down, since these rely on the assumption that the function is smooth in a small neighborhood of the point at which the discrete difference is taken. Instead of seeking solutions in a *pointwise* sense, the finite-volume approach seeks *globally defined solutions* of the integral form (9.2), which is more fundamental, represented in the form of discrete cell averages.

To develop a finite-volume discretization of (9.1), we first define the sliding average,

$$\bar{u}(x, t) = \frac{1}{\Delta x} \int_{x - \frac{1}{2}\Delta x}^{x + \Delta x/2} u(\xi, t) d\xi. \quad (9.6)$$

We then associate these sliding averages with a set of grid cells, $\{[x_{i-1/2}, x_{i+1/2}]\}$, where $x_{i\pm 1/2} = x_i \pm \frac{1}{2}\Delta x_i$, and set $u_i(t) = \bar{u}(x_i, t)$. By inserting $u_i(t)$ into the integral form of the conservation law (9.2), we obtain a semi-discrete version of (9.1),

$$\frac{d\bar{u}_i}{dt} = \frac{1}{\Delta x_i} [f(u(x_{i-1/2}, t)) - f(u(x_{i+1/2}, t))]. \quad (9.7)$$

Alternatively, we may define $u_i^n = \bar{u}_i(t_n)$ for a set of discrete times t_n and then integrate (9.7) from t_n to t_{n+1} to derive a fully-discrete version of (9.1),

$$u_i^{n+1} - u_i^n = \frac{1}{\Delta x} \int_{t_n}^{t_{n+1}} f(u(x_{i-1/2}, t)) dt - \frac{1}{\Delta x} \int_{t_n}^{t_{n+1}} f(u(x_{i+1/2}, t)) dt. \quad (9.8)$$

While these (9.7) and (9.8) tell us how the unknown cell averages $u_i(t)$ and u_i^n evolve in time, they cannot be used directly to compute these cell averages since we do not know the *point values* $u(x_{i\pm 1/2}, t)$. To obtain these point values, we will need to make additional assumptions and approximations. However, (9.8) suggests that a viable numerical method for (9.1) should be of the form

$$u_i^{n+1} = u_i^n - r_i^n (F_{i+1/2}^n - F_{i-1/2}^n), \quad (9.9)$$

where $r_i^n = \Delta t^n / \Delta x_i$ and $F_{i\pm 1/2}^n$ is some approximation to the average flux over each cell interface,

$$F_{i\pm 1/2}^n \approx \frac{1}{\Delta t^n} \int_{t_n}^{t_{n+1}} f(u(x_{i\pm 1/2}, t)) dt. \quad (9.10)$$

In Section 8.4 we showed that information propagates at a finite speed along so-called characteristics, which implies that the flux integral (9.10) will only depend on the solution $u(x, t_n)$ in a local neighborhood of the interface $x_{i\pm 1/2}$. This means, in turn, that $F_{i\pm 1/2}$ can be approximated in terms of a small collection of neighboring cell averages, i.e., $F_{i+1/2} = F(u_{i-p}^n, \dots, u_{i+q}^n)$, which we alternatively will write as $F_{i\pm 1/2} = F(u^n; i \pm \frac{1}{2})$.

Any numerical method written on the form (9.9) will be *conservative*. To see this, we multiply (9.9) by Δx_i and sum over i . The flux terms will cancel in pairs and we are left with

$$\sum_{i=-M}^N u_i^{n+1} \Delta x_i = \sum_{i=-M}^N u_i^n \Delta x_i - \Delta t^n (F_{N+1/2}^n - F_{-M-1/2}^n).$$

From this, it follows that the method is conservative on a finite domain since the accumulation inside any interval balances the sum of the flux across the interval edges. To see that the same is true in an infinite domain, we must make some additional assumptions. If the initial solution $u_0(x)$ has bounded support, it follows that the two flux terms will cancel if we choose M, N sufficiently large. Hence, the scheme (9.9) is conservative in the sense that

$$\sum_i u_i^{n+1} \Delta x_i = \sum_i u_i^n \Delta x_i = \dots = \int u_0(x) dx.$$

9.3 A few classical schemes

In this section, we will go through a number of classical schemes. Although these were developed several decades ago, they are still in use and provide important insight into the numerical solution of hyperbolic equation.

9.3.1 Centered schemes

To be able to evolve the cell averages $u_i(t)$ and u_i^n according to (9.7) and (9.9), respectively, we must make some assumption that enable us to *reconstruct* the point values $u(x_{i\pm 1/2}, t)$. The simplest possible choice is to assume that $u(x, t) = u_i(t)$ inside each grid cell. To get values at the grid interfaces $x_{i\pm 1/2}$ we can simply average the flux values on opposite sides of the grid interface, i.e., set

$$F_{i\pm 1/2}^n = \frac{1}{2} [f(u_{i\pm 1}^n) + f(u_i^n)]. \quad (9.11)$$

This approximation is referred to as a *centered approximation*. Unfortunately, this gives a scheme that is notoriously unstable. To stabilize, we can add

artificial diffusion, $\frac{\Delta x^2}{\Delta t} \partial_x^2 u$. Notice, however, that by doing this, it is no longer possible to go back to the semi-discrete form (9.7), since the artificial diffusion will blow up in the limit $\Delta t \rightarrow 0$. If we discretize the artificial diffusion standard centered differences, we get the classical Lax–Friedrichs scheme

$$u_i^{n+1} = \frac{1}{2} \left(u_{i+1}^n + u_{i-1}^n \right) - \frac{1}{2} r \left[f(u_{i+1}^n) - f(u_{i-1}^n) \right]. \quad (9.12)$$

Alternatively, the scheme can be written in conservative form (9.9) using the numerical flux

$$F(u^n; i + 1/2) = \frac{1}{2r} (u_i^n - u_{i+1}^n) + \frac{1}{2} [f(u_i^n) + f(u_{i+1}^n)]. \quad (9.13)$$

To ensure stability, we have to impose a restriction on the time-step through a *CFL condition*, named after Courant, Friedrichs, and Lewy, who wrote one of the first papers on finite difference methods in 1928 [65]. The CFL condition states that the true domain of dependence for the PDE (9.1) should be contained in the domain of dependence for (9.9). For the Lax–Friedrichs scheme, this means that

$$\frac{\Delta t}{\Delta x} \max_u |f'(u)| \leq 1. \quad (9.14)$$

Under this condition, the scheme is very robust and will always converge, albeit painstakingly slow in many cases. The robustness is largely due to the added numerical diffusion, which tends to smear discontinuities. As an illustration, consider a stationary discontinuity satisfying $\partial_t u = 0$. Here, the approximate solution in each cell is defined as the arithmetic average of the cell averages in the neighboring cells, i.e., $u_i^{n+1} = \frac{1}{2} (u_{i+1}^n + u_{i-1}^n)$. From this, we see that the scheme will smear the discontinuity one cell in each direction per time step. By using a formal Taylor expansion on a single time step, we can show that the scheme has a truncation error of order two. In practice, we are more interested in the error at a fixed time, which we will use an increasing number of time steps to reach as $\Delta x \rightarrow 0$ because of (9.14). This means that we must divide with Δt and hence the formal error of the scheme will be $\mathcal{O}(\Delta x)$ as $\Delta x \rightarrow 0$, and we say that the scheme is *formally first order accurate*.

Better accuracy can be obtained if we make a more accurate approximation to the integral that defines $F_{i\pm 1/2}^n$. Instead of evaluating the integral at the endpoint t_n , we can evaluate it at the midpoint $t_{n+1/2} = t_n + \frac{1}{2}\Delta t$. One can show that the corresponding point values can be *predicted* with acceptable accuracy by using the Lax–Friedrichs scheme on a grid with half the grid spacing. Altogether, this gives a second-order, predictor-corrector scheme called the the Richtmyer two-step Lax–Wendroff method [146]

$$\begin{aligned} u_{i+1/2}^{n+1/2} &= \frac{1}{2} (u_i^n + u_{i+1}^n) - r [f(u_{i+1}^n) - f(u_i^n)], \\ u_i^{n+1} &= u_i - r [f(u_{i+1/2}^{n+1/2}) - f(u_{i-1/2}^{n+1/2})], \end{aligned} \quad (9.15)$$

which is stable under the same CFL condition (9.14) as the Lax–Friedrichs scheme. The corresponding numerical flux reads

$$F(u^n; i + 1/2) = f\left(\frac{1}{2}(u_i^n + u_{i+1}^n) - \frac{1}{2}r[f(u_{i+1}^n) - f(u_i^n)]\right). \quad (9.16)$$

Both the Lax–Friedrichs and the Lax–Wendroff scheme can either be interpreted as a finite-difference or as a finite-volume scheme, and by only looking on the formulas written in terms of u_i^n , it is not possible to determine which formulation is in use. However, the underlying principles are fundamentally different. Finite-difference methods evolve a discrete set of point values by using discrete differences to approximate the differential operators in (9.1). Finite-volume methods evolve *globally defined solutions* as given by (9.2) and realize them in terms of a discrete set of cell averages. This latter perspective is the key to modern so-called high-resolution methods, which have proved to be very successful. We will come back to this in Section 9.5.

Let us now see how we can implement these two schemes compactly in MATLAB. When computing on a bounded domain, one generally has to modify the stencil of the scheme in the cells next to the domain boundary to account for boundary conditions. A widely used trick to avoid this is to pad the domain with a layer of *ghost cells*, i.e., extra cells that are not really part of the domain and whose states can be set manually to impose the desired boundary conditions.

```
function u=lxf(u0,cfl,dx,T,flux,df,boundary)
u = u0; t = 0.0;
dt = cfl*dx/max(abs(df(u0)));
i = 2:numel(u0)-1; % do not compute on the ghost cells
while (t < T)
    if t+dt > T, dt = T-t; end
    t=t+dt;
    u = boundary(u); % set values in the ghost cells
    f = flux(u);
    u(i) = 0.5*(u(i+1)+u(i-1)) - 0.5*dt/dx*(f(i+1)-f(i-1));
    dt = cfl*dx/max(abs(df(u)));
end
```

Here, u_0 gives the initial data, cfl and dx are the CFL number and the spatial discretization parameter, T is the desired end time, and $flux$, df , and $boundary$ are function handles to functions that compute the flux f and its derivative and set appropriate states in the ghost cells. Notice, in particular, how we use the vector i to avoid computing in the ghost cells that pad our domain. To get the Lax–Wendroff scheme, we introduce an auxiliary array U and then simply replace the second last line by,

```
U(i) = 0.5*(u(i)+u(i+1)) - 0.5*r*(f(i+1)-f(i));
U = boundary(U); f = flux(U);
u(i) = u(i) - r*(f(i)-f(i-1));
```

9.3.2 Upwind or Godunov schemes

The centered schemes introduced above are black-box schemes that can be utilized without any particular knowledge about the flux function apart from an estimate of its maximum and minimum derivative (or eigenvalues for systems of equations). The reason for this is that these schemes utilize information from both sides of the grid interface when computing approximations to the flux integral in (9.10). However, as we saw in Section 8.4, this is general not necessary if we know the structure of the flux function. If $f'(u) > 0$, for instance, we know that all characteristics are positive so that we only need to use the solution from the left side of the grid interface to evaluate the integral in (9.10). With a constant reconstruction inside each cell, this means that $F_{i+1/2} = f(u_i^n)$ is the *exact* value of the integral, provided that the solution at time t_n is constant and equal u_i^n in grid cell i . The resulting scheme,

$$u_i^{n+1} = u_i^n - r[f(u_i^n) - f(u_{i-1}^n)] \quad (9.17)$$

is called an *upwind scheme*. Similarly, if $f'(u) \leq 0$, the upwind scheme takes the form

$$u_i^{n+1} = u_i^n - r[f(u_{i+1}^n) - f(u_i^n)]. \quad (9.18)$$

In either case, the upwind scheme is a two-point scheme that instead of using a centered difference uses a one-sided difference in the *upwind* direction, i.e., in the direction from which the waves (or information) is coming. This type of upwind differencing is the underlying design principle for so-called *Godunov schemes*, named after the author of the seminal paper [94]. If we, as above, assume a constant reconstruction so that $u(x, t^n) \equiv u_i^n$ inside grid cell number i , the evolution of $u(x, t)$ can be decomposed into a set of local *Riemann problems*

$$\partial_t v + \partial_x f(v) = 0, \quad v(x, 0) = \begin{cases} u_i^n, & x < x_{i+1/2}, \\ u_{i+1}^n, & x \geq x_{i+1/2}, \end{cases} \quad (9.19)$$

which are of the same type as those we have already accounted in Section 8.4. Each of these Riemann problems admits a self-similar solution $v_{i+1/2}(x/t)$, which can be constructed by introducing a local convex or concave envelope¹ $f_c(u; u_i^n, u_{i+1}^n)$ of the flux function f over the interval $[u_i^n, u_{i+1}^n]$, as discussed in (8.59) on page 267. The cell averages can then be correctly evolved a time step Δt forward in time by (9.9) if we use $v_{i\pm 1/2}(0)$ —or a good approximation thereof—to evaluate f in the flux integral (9.10). The Riemann fan will generally expand in each direction at a constant speed, and using $v_{i\pm 1/2}(0)$ to evaluate the integrand of (9.10) is only correct until the first wave from the neighboring Riemann problem reaches the interface. This implies a time-step restriction of the form

¹ The notation $f(\cdot; a, b)$ signifies that the functional form of the function f depends on the parameters a and b .

$$\max_{i,n} |f'_c(\cdot; u_i^n, u_{i+1}^n)| \frac{\Delta t^n}{\Delta x_i} \leq 1, \quad (9.20)$$

which we recognize as a sharper version of the CLF condition (9.14). The resulting scheme is formally first-order accurate, but will generally be much less diffusive than the centered Lax–Friedrichs scheme.

Let us now go back to the general transport equation (8.34), which is hyperbolic if $P'_c \equiv 0$. Here, the fractional flow function $f(S)$ has a characteristic S -shape and its derivative is therefore always positive. This means that if gravity forces are negligible, like in the horizontal displacements we studied in Section 8.4.1, the correct Godunov scheme would be either (9.17) or (9.18), depending upon the sign of the flux \vec{v} . For the gravity segregation problem considered in Section 8.4.2, on the other hand, the flux function $g(S) = \lambda_w \lambda_n / (\lambda_w + \lambda_n)$ has derivatives of both signs and we would generally have to solve the Riemann problem (9.19) like we did in Figure 8.14 on page 271 in order to apply the Godunov scheme. The same is generally true for flow that involves a combination of viscous and gravity forces, as illustrated in Figure 8.17 on page 275. Having to solve a Riemann problem each time we need to evaluate the flux $F_{i\pm 1/2}^n$ may seem rather cumbersome, and unfortunately there is a rather elegant and physically motivated fix to this. Instead of upwinding the flux itself, we can use the phase fluxes to evaluate the relative mobilities that enter both the S -shaped viscous flux $f_w(S)$ and the bell-shaped gravity flux $\lambda_n(S)f_w(S)$ in the upstream direction

$$\lambda_\alpha(S)^u|_{x_{i+1/2}} = \begin{cases} \lambda_\alpha(S_i^n), & \text{if } v_\alpha > 0, \\ \lambda_\alpha(S_{i+1}^n), & \text{otherwise.} \end{cases} \quad (9.21)$$

Here, we have used the phase flux v_α which in the multidimensional case can be computed as $(\nabla p_\alpha - g\rho_\alpha \nabla z) \cdot \vec{n}_i$, where \vec{n}_i is the normal to the interface.

The resulting method, called the explicit *upstream mobility-weighting* scheme, is often the method of choice for porous media application because of its accuracy and simplicity. In [43] it was shown that method also satisfies the mathematical properties that are necessary to get a consistent and convergent scheme. We will come back to these properties in Section 9.4.

9.3.3 Comparison of centered and upwind schemes

The classical schemes introduced above all have their weaknesses and strengths. In this section we will consider two different cases that highlights these. Complete source codes for the two examples can be found in the `nummet` directory of the `book` module.

Example 9.1. As our first test case, we consider linear advection on the unit interval with periodic boundary conditions

$$u_t + u_x = 0, \quad u(x, 0) = u_0(x), \quad u(0, t) = u(1, t).$$

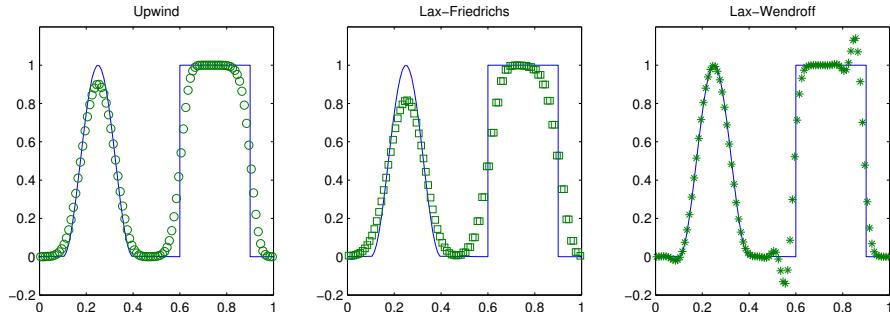


Fig. 9.1. Approximate solutions at time $t = 20.0$ for the linear advection equation $u_t + u_x = 0$ with periodic boundary conditions computed by three classical schemes on a grid with 100 cells and CFL number 0.995.

The advantage of using periodic data is that we know that the exact solution at time $t = n$ for an integer number n equals $u_0(x)$. As initial data $u_0(x)$ we choose a combination of a smooth squared sine wave and a double step function,

$$u(x, 0) = \sin^2\left(\frac{x - 0.1}{0.3}\pi\right)\chi_{[0.1, 0.4]}(x) + \chi_{[0.6, 0.9]}(x).$$

This initial data contains both a smooth wave and two discontinuous waves. Using the MATLAB solvers introduce above, we can set up the case as follows:

```

dx = 1/100;
x = -.5*dx:dx:1+.5*dx;
u0 = sin((x-.1)*pi/.3).^2.*double(x>=.1 & x<=.4);
u0((x<.9) & (x>.6)) = 1;

f = @u; u; % flux function
df = @u; 0*u+1;
uf = lxf(u0, .995, dx, 20, f, df, @periodic);

```

where the function `periodic` simply sets `u(1)=u(end-1)` and `u(end)=u(2)`.

Figure 9.1 shows approximate solutions after twenty periods ($t = 20$) computed by the upwind, Lax–Friedrichs, and Lax–Wendroff methods on a grid with 100 cells for $\Delta t = 0.995\Delta x$. Both the centered schemes clearly give unacceptable resolution of the solution profile. The first-order Lax–Friedrichs scheme smears both the smooth and the discontinuous part of the advected profile. The second-order Lax–Wendroff scheme preserves the smooth profile quite accurately, but introduces spurious oscillations at the two discontinuities. This behavior is representative for classical schemes. The upwind method represents a reasonable compromise in the sense that it does not smear the smooth wave and the discontinuities as much as the Lax–Friedrichs scheme and resolves the discontinuities more stable than the Lax–Wendroff scheme.

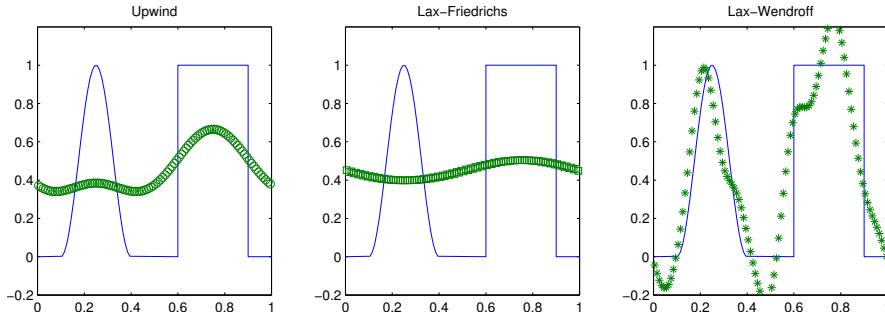


Fig. 9.2. Approximate solutions at time $t = 5.0$ for the linear advection equation $u_t + u_x = 0$ with periodic boundary conditions computed by three classical schemes on a grid with 100 cells and CFL number 0.5.

The simple advection equation seldom appears in the modelling of porous media and hence has limited direct interest. However, the equation can serve as a conceptual model for more complex cases. As we saw in Figures 8.9 and 8.10 on page 267, a displacement front will be self-sharpening in the sense that if the leading discontinuity is smeared, the smeared states behind the displacement front will tend to travel faster than the discontinuity (since they have characteristics that point into the front) and hence 'catch up' with it. Likewise, smeared values ahead of the front tend to travel slower and hence be overrun by the front, since these states have characteristics that point into it. For linear waves, the characteristics of any smeared state will run parallel with the discontinuity and hence there will be no self-sharpening to counteract the numerical dissipation that is inherent in any numerical scheme. Linear waves are therefore more susceptible to numerical dissipation than nonlinear shock waves.

In water-based EOR methods, for instance, active chemical or biological substances are added to modify the physical properties of the fluids or/and the porous media at the interface between oil and water. The resulting displacement processes are governed by complex interplays between the transport of chemical substances, and how these substances affect the flow by changing the properties of the fluids and the surrounding rock. These changes in properties are often very nonlinear and highly sensitive to threshold parameters that determine sharp transitions between regions of very different behavior. The transport of chemical substances is largely linear and hence more affected by numerical diffusion. A simulation may therefore fail to resolve the local displacement process if the discontinuity of active components is smeared out or contains overshoots. As a result, unresolved simulation can therefore lead to misleading predictions of injectivity and recovery profiles.

For many systems, the linear waves travel slower than the leading shock wave, and hence will be computed with an effective CFL number that is smaller than the optimal choice of 1.0. In Figure 9.2, we have repeated the

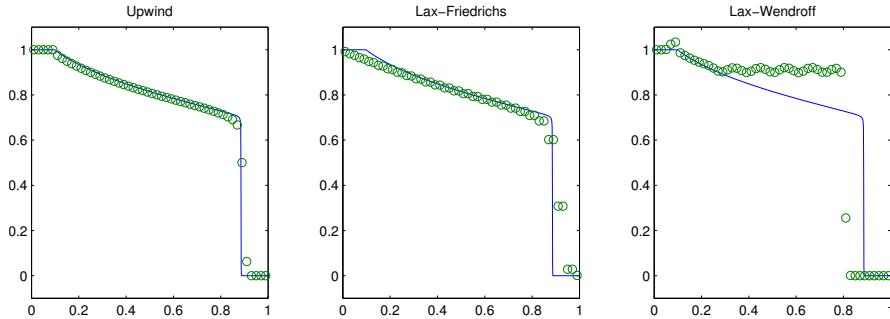


Fig. 9.3. Approximate solutions at time $t = 0.65$ for the Buckley–Leverett problem on a grid with 50 cells computed by three classical schemes.

same setup as above, except that we now report the solution at time $t = 5$ and use a CFL number of 0.5, assuming that the linear wave travels at half the speed of a leading shock wave. In this case, neither of the schemes can be said to compute a satisfactory solution; whereas the two first-order schemes have smeared both the continuous and the discontinuous bumps almost beyond recognition, the second-order scheme has introduced large oscillations at the two discontinuities that have propagated so that they have polluted the smooth bump. At time $t = 20$, the Lax–Friedrichs solution has more or less approached a constant state, the upwind solution has barely visible bumps, while the Lax–Wendroff solution has deformed bumps that travel slower than they ought to. This is admittedly a somewhat extreme case, but it points out very well some of the potential pitfalls of classical schemes.

In the next example, we consider a setup that has direct relevance to two-phase flow.

Example 9.2. Let us revisit the classical Buckley–Leverett profile studied in Example 1, i.e., consider the following equation,

$$S_t + \left(\frac{S^2}{S^2 + (1-S)^2} \right)_x = 0, \quad S(0, t) = 1, \quad S(x, 0) = \begin{cases} 1, & x < 0.1, \\ 0, & x \geq 0.1 \end{cases}$$

Figure 9.3 shows the solution at time $t = 0.65$ computed by the three schemes. The two first-order schemes manage to resolve both the leading shock wave and the trailing rarefaction wave with reasonable accuracy, but we observe that the centered Lax–Friedrichs scheme smears the shock and the kink more than the upwind scheme, as expected. The second-order Lax–Wendroff scheme does not resolve the composite wave correctly. The reason is the following. As we saw for the linear advection case, the scheme will tend to introduce overshoots at the leading discontinuity, and the only way to maintain a mass-conservative solution is if the numerically computed wave (with overshoots) travels slower than the true wave. Likewise, the scheme introduces an overshoot at the kink where the solution has discontinuous derivative. These are a serious deficiencies of this scheme.

COMPUTER EXERCISES:

52. Try to set $\Delta x = \Delta t$ so that the CFL number equals one exactly for the periodic advection problem in Example 9.1. Can you explain what you observe? Why is using a CFL number identical to one not that interesting from a practical point of view?
53. Implement a Godunov solver that works correctly for strictly convex or strictly concave flux functions. Hint: try to find a formula that gives the correct self-similar Riemann solution along the line $x/t = 0$.
54. Implement the single-point upstream mobility-weighting scheme (9.21) and use it to simulate the 1D cases discussed in Examples 8.2, 8.3, and 8.4.
55. Extend the schemes introduced above to the two-dimensional conservation law $u_t + f(u)_x + g(u)_y = 0$, and implement a solver that can run a 2D analogue of the periodic advection problem in Example 9.1. Try to make the changes in the existing solver as small as possible. Hint: In 2D, the Lax–Fridrichs scheme reads,

$$\begin{aligned} u_{ij}^{n+1} &= \frac{1}{4}(u_{i+1,j}^n + u_{i-1,j}^n + u_{i,j+1}^n + u_{i,j-1}^n) \\ &\quad - \frac{1}{2}r[f(u_{i+1,j}^n) - f(u_{i-1,j}^n)] - \frac{1}{2}r[g(u_{i,j+1}^n) - g(u_{i,j-1}^n)]. \end{aligned}$$

9.3.4 Implicit schemes

So far, we have only considered schemes that have relied on explicit temporal discretization since this is the most common approach in the literature on numerical methods for hyperbolic conservation laws. In principle, we could also have used *implicit* temporal discretization, as we have already encountered in Chapter 7. Going back to (9.8) on page 282, we could approximate the integral using point values at t_{n+1} rather than at time t_n , which would give us a numerical method of the form,

$$u_i^{n+1} + r_i^n(F_{i+1/2}^{n+1} - F_{i-1/2}^{n+1}) = u_i^n. \quad (9.22)$$

To get a specific scheme, we must specify how the cell averages $u_{i-p}^{n+1}, \dots, u_{i+q}^{n+1}$ should be used to evaluate the numerical flux $F_{i+1/2}^{n+1}$. The standard approach in reservoir simulation is to use the upstream mobility-weighting approximation from (9.21) with S^n replaced by S^{n+1} .

In the explicit method (9.9), the flux terms were written on the right-hand side to signify that they are given in terms of the known solution u_i^n . In (9.22), the numerical flux terms appear on the left-hand side to signify that they depend on the unknown cell averages u_i^{n+1} . Equation (9.22) hence represents a coupled system of *nonlinear* discrete equations, which typically must be solved using a Newton–Raphson method as discussed in Section 7.1. This means that a computing a single time step of an implicit method will

typically be significantly more costly than computing an explicit update, since the latter will involve fewer evaluations of the flux function and its derivatives. On the other hand, the fully-implicit discretization (9.22) is *unconditionally stable* in the sense that there is no stability condition that limits the size of the time step. In principle, this means that the additional computational cost can be offset by using much larger time steps. In practice, however, the sizes of the time steps one can use are limited by numerical errors (as we will see shortly) and by the convergence of the numerical method used to solve the nonlinear system. For Newton–Raphson type methods, in particular, the convergence may be quite challenging if the flux function contains inflection points/lines that separate regions of different convexity, see e.g., [119, 254] for a more detailed discussion.

For a 1D horizontal displacement, we have already seen that the flux function has a characteristic *S*-shape with positive derivatives. Like in the explicit case, we can therefore use a one-sided values to evaluate the flux integrals, giving the following scheme,

$$u_i^{n+1} + r_i^n [f(u_i^{n+1}) - f(u_{i-1}^{n+1})] = u_i^n.$$

The result is a triangular nonlinear system with one nonzero band below the diagonal. For cases like the Buckley–Leverett displacement considered in Example 9.2, we can therefore solve the nonlinear system very efficiently using a substitution method. That is, we start at the cell next to the inflow boundary, in which the flow problem reads

$$u_1^{n+1} + r_1^n f(u_1^{n+1}) = u_1^n + r_1^n f_L,$$

where f_L is the known inflow. This single nonlinear equation can be solved robustly for an arbitrarily large Δt by using a classical bracketing method that tracks the end points of an interval containing the unknown root. Once u_1^{n+1} has been computed, we can move to the next cell, and solve a similar scalar equation,

$$u_2^{n+1} + r_2^n f(u_2^{n+1}) = u_1^n + r_2^n f(u_1^{n+1}),$$

and so on. The same strategy can, in fact, be applied to the multidimensional case under certain assumptions that guarantee a system having a similar concurrent flow property, see [186] for more details.

Example 9.3. Let us revisit the setup from Example 9.2 and compare the explicit (9.9) and the implicit (9.22) discretizations. Instead of implementing the 1D implicit scheme, we will rely on the general transport solvers from the `incomp` module in MRST, which will be discussed in more detail in the next chapter. Figure 9.4 shows solutions computed by the explicit scheme with a unit CFL number and by the implicit scheme with CFL number 1 and 10. At a unit CFL number, the explicit scheme gives a somewhat sharper resolution of the displacement front than the implicit scheme. When the CFL

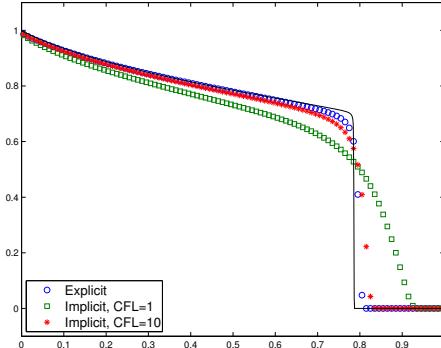


Fig. 9.4. Approximate solutions at time $t = 0.65$ for the Buckley–Leverett problem on a grid with 100 cells computed by the single-point upstream mobility-weighting scheme with explicit and implicit time discretization.

number is increased ten times for the implicit scheme, the numerical smearing is significant and will lead to the prediction of a (much) too early water breakthrough.

To explain the difference in numerical smearing, let us do some simple numerical analysis. However, rather than studying the nonlinear Buckley–Leverett model, we will go back to the simple advection equation, $u_t + au_x = 0$, and consider the schemes above as finite-difference schemes rather than finite-volume schemes. The classical way of analyzing numerical schemes is to first compute the truncation error, which is obtained by inserting the exact solution into the numerical scheme, assume a smooth solution, and use Taylor expansions to express the various terms around a common point (x, t) . In addition, we will also use the fact that $u_t = -au_x$ and that Δt and Δx are proportional. Starting with the explicit scheme,

$$\begin{aligned} 0 &= \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} + a \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x} \\ &= u_t(x, t) + \frac{1}{2}\Delta t u_{tt}(x, t) + \mathcal{O}(\Delta t^2) + au_x(x, t) - \frac{1}{2}a\Delta x u_{xx}(x, t) + \mathcal{O}(\Delta x^2) \\ &= u_t + \frac{1}{2}\Delta t a^2 u_{xx} + \mathcal{O}(\Delta t^2) + au_x - \frac{1}{2}a\Delta x u_{xx} + \mathcal{O}(\Delta x^2) \\ &= u_t + au_x - \frac{1}{2}a[\Delta x - a\Delta t]u_{xx} + \mathcal{O}(\Delta x^2) \end{aligned}$$

In other words, the explicit scheme will up to order $\mathcal{O}(\Delta x^2)$ solve a *parabolic* equation. This equation is only well-posed if the term in front of the second-order term is negative, implying that $a\Delta t/\Delta x \leq 1$, which is exactly the same requirement as in the CFL condition. We also notice that the higher the CFL number we use, the less numerical diffusion we will introduce in the approximate solution and in the special case that $a\Delta t = \Delta x$, the scheme will have no numerical diffusion.

For the implicit scheme we only need to expand the u_{i-1}^{n+1} term since we have already compute the expansion for u_i^{n+1} for the explicit scheme,

$$\begin{aligned} u_{i-1}^{n+1} &= u(x - \Delta x, t + \Delta t) \\ &= u(x - \Delta x, t) - a\Delta t u_x(x - \Delta x, t) + \frac{1}{2}(a\Delta t)^2 u_{xx}(x - \Delta x, t) + \mathcal{O}(\Delta t^3) \\ &= u - \Delta x u_x + \frac{1}{2}\Delta x^2 u_{xx} + \mathcal{O}(\Delta x^3) - a\Delta t \partial_x [u - \Delta x u_x + \mathcal{O}(\Delta x^2)] \\ &\quad + \frac{1}{2}(a\Delta t)^2 \partial_x^2 [u + \mathcal{O}(\Delta x)] + \mathcal{O}(\Delta t^3) \end{aligned}$$

Collecting terms, we have that

$$\begin{aligned} 0 &= \frac{u(x, t + \Delta t) - u(x, t)}{\Delta t} + a \frac{u(x, t) - u(x - \Delta x, t)}{\Delta x} \\ &= u_t + au_x - \frac{1}{2}a[\Delta x + a\Delta t]u_{xx} + \mathcal{O}(\Delta x^2). \end{aligned}$$

This means that the implicit scheme also will solve a parabolic equation with accuracy $\mathcal{O}(\Delta x^2)$. In this case, the coefficient in front of the u_{xx} term is always negative and the scheme will therefore be stable regardless of the choice of Δt . Notice, however, this also means that the numerical diffusion *increases* with increasing time steps.

A similar analysis can also be conducted in the nonlinear case and will lead to similar conclusions. This in turn implies an interesting observation if we look back at our discussion of linear waves on page 289. For an explicit scheme, slow waves will be computed with *more* numerical diffusion than fast waves. For an implicit scheme, on the other hand, slow waves will be computed with *less* numerical diffusion than fast waves. This implies that explicit schemes are best suited for systems having relatively small differences in wave speeds, whereas implicit schemes are better suited for systems with large differences in wave speeds. In a typical reservoir setting, there will be large wave speeds in the near-well region, medium wave speeds in high-flow regions away from the wells, and low wave speeds near and in stagnant zones. It is therefore reasonable to expect that implicit schemes will be best suited for simulation of real reservoir systems. However, explicit schemes are still used, mainly because of their simplicity and in connection with higher-order discretizations, which we will discuss in Section 9.5 on page 297.

COMPUTER EXERCISES:

- 56. Implement the implicit upwind scheme in 1D and verify against the solver from the `incomp` module.
- 57. Use the single-point upstream mobility-weighting scheme (9.21) to extend your implicit solver to also account for counter-current flow and use it to simulate the 1D cases discussed in Examples 8.2, 8.3, and 8.4. Compare with the corresponding explicit scheme.

9.4 Convergence of conservative methods

How can we be sure that the numerical schemes we have introduced above compute correct approximations to the governing partial differential equation? And likewise, how can we be sure that our scheme converges to the correct solution as Δt and Δx tend to zero? In this section we will try to answer these two questions. The resulting numerical analysis will provide us with useful insight that will be used when developing so-called high-resolution discretizations in Section 9.5. In the following, we will use the notation $u_{\Delta t}(x, t)$ to define a piecewise continuous reconstruction of $u(x, t)$ based on the cell-averaged values u_i^n (or a piecewise continuous interpolation in the case of a finite-difference scheme). The simplest such reconstruction is to assume that $u(x, t) = u_i^n$ for $(x, t) \in [x_{i-1/2}, x_{i+1/2}] \times [t_n, t_{n+1}]$.

In the previous section, we showed how to derive the *truncation error* error of a numerical scheme. For the explicit scheme (9.9), this error is defined as

$$T_{\Delta t} = \frac{1}{\Delta t} \left[u(x, t+\Delta t) - u(x, t) + r[F(u(x, t); i+\frac{1}{2}) - F(u(x, t); i-\frac{1}{2})] \right]. \quad (9.23)$$

In classical numerical analysis, one then shows that $T_{\Delta t} = \mathcal{O}(\Delta t^m)$ for some $m > 0$, where the exponent m is said to be the order of the scheme. This means that the explicit scheme (9.9) and the (9.22) are both first order methods. Unfortunately, hyperbolic equations seldom admit solutions in the classical sense and hence we cannot use the pointwise truncation error to establish convergence. Instead, this analysis is used to establish the *formal order* of a method, i.e., the order the method would converge with for smooth solutions.

Instead of using pointwise errors, we can study the convergence of $u_{\Delta t}(x, t)$ in some function space and measure the error in an appropriate norm defined on this function space. For scalar conservation laws, it turns out that the correct function space is L^1 , which consists of all functions v having a bounded L^1 norm, $\|v\|_1 = \int |v| d\Omega$. The reason for using L^1 is that the weak solutions have two appealing properties in this space. First of all, any entropy-weak solution is L^1 *contractive* in the sense that,

$$\|u(\cdot, t)\|_1 \leq \|u(\cdot, s)\|_1, \quad \text{for } t \geq s. \quad (9.24)$$

Likewise, the total variation of the exact solution of a scalar conservation law is non-increasing with time,

$$\text{TV}(u(\cdot, t)) \leq \text{TV}(u(\cdot, s)), \quad \text{for } t \geq s. \quad (9.25)$$

The total variation of a function $v \in L^1(\mathbf{R})$ is defined as

$$\text{TV}(u) = \limsup_{\epsilon \rightarrow 0} \frac{1}{\epsilon} \int_{-\infty}^{\infty} |v(x + \epsilon) - v(x)| dx. \quad (9.26)$$

From functional analysis, we have that the set of functions with bounded total variation is compact L^1 ; that is, the set contains all its limit points and

is bounded in the sense that all functions in this set lie a finite distance from each other.

Working in L^1 , we thus say that $u_{\Delta t}(x, t)$ converges to $u(x, t)$ if

$$\int_0^t \|u_{\Delta t}(\cdot, t) - u(\cdot, t)\|_{L^1(\mathbb{R})} dt \rightarrow 0, \text{ as } \Delta t \rightarrow 0.$$

So, how can we guarantee that this condition is satisfied? The famous Lax–Wendroff theorem [146] states that if an sequence of approximate solutions computed by a *conservative and consistent* scheme converges to a limit, then this limit is a weak solution. We have already verified that any scheme written on the form (9.9) is conservative. Similarly, the scheme (9.9) is said to be consistent if

$$\varphi_t + f(\varphi)_x - \left[\frac{\varphi(x, t+\Delta t) - \varphi(x, t)}{\Delta t} - \frac{F(\varphi; i+\frac{1}{2}) - F(\varphi; i-\frac{1}{2})}{\Delta x} \right] \rightarrow 0$$

as the discretization parameters Δx and Δt tend to zero. This is guaranteed if the numerical flux function F satisfies the following property,

$$F(v, \dots, v) = f(v). \quad (9.27)$$

In addition, it is common to require that the numerical method is Lipschitz continuous, i.e., that there is a constant L such that

$$|F(u_{i-p}, \dots, u_{i+q}) - f(u)| \leq L \max(|u_{i-p} - u|, \dots, |u_{i+q} - u|).$$

For linear equations, the *Lax equivalence theorem*, which is also sometimes called the *fundamental theorem of numerical analysis*, states that a scheme is convergent if and only if it is stable and consistent. A scheme is stable if the errors introduced at a time step do not grow (too fast) in time. Stability is ensured through the CFL condition for explicit schemes, whereas the implicit schemes (9.22) are stable by construction.

For nonlinear equations, similar results hold if we can prove that the numerical scheme is contractive in some appropriate norm since this guarantees that the sequence of cell averages is compact² and thus convergent. We already know that the total variation of any exact solution to the conservation law is non-increasing with time, see (9.25), and that functions of bounded variations form a compact set in L^1 . It is therefore natural to require the same of our approximate solutions, i.e.,

$$\text{TV}(u_i^{n+1}) \leq \text{TV}(u_i^n), \quad (9.28)$$

where the total variation for a piecewise constant function $v = v_i$ in $[x_{i-1/2}, x_{i+1/2}]$ is defined as,

² A sequence $\{s_1, s_2, \dots\}$ of elements in a space S is compact if it contains a subsequence that converges to an element in S .

$$\text{TV}(v) = \sum_i |v_i - v_{i-1}|.$$

Any scheme satisfying (9.28) is called a *total variation diminishing* method, commonly abbreviated as a TVD-method. For conservative and consistent methods, we have the following list of implications among provable properties:

- Any monotone method is L^1 contractive, i.e., $\|u_{\Delta t}(\cdot, t)\|_1 \leq \|u_{\Delta t}(\cdot, 0)\|_1$. A method is monotone if

$$u_i^n \geq v_i^n \quad i \in \mathbb{Z}, \quad \Rightarrow \quad u_i^{n+1} \geq v_i^{n+1}, \quad i \in \mathbb{Z}.$$

- Any L^1 contractive method is TVD.
- Any TVD method is monotonicity preserving, which means that if the initial data u_i^0 is monotone then so is u_i^n for any n .

In practice, it is typically simplest to prove that a method is monotone. Unfortunately, any monotone method is at most first-order accurate, and to design methods of higher-order accuracy it is therefore common to use the weaker TVD assumption.

Using the Lax-Wendroff and compactness theory reviewed above, we can verify that a scheme converges to a weak solution of the conservation law. However, this does not tell us whether the limit is the correct entropy weak solution or not. To this end, one can show that if the scheme (9.9) satisfies a discrete variant of (9.4)

$$\eta(u_i^{n+1}) \leq \eta(u_i^n) - \lambda(\Psi_{i+1/2}^n - \Psi_{i-1/2}^n), \quad (9.29)$$

then the limiting weak solution is the entropy-weak solution. Here Ψ is a numerical entropy flux that must be consistent with the entropy flux ψ in the same way as the numerical flux F should be consistent with f .

9.5 High-resolution schemes

So-called high-resolution schemes are designed to have second or higher order spatial accuracy on smooth parts of the solution and at the same time be non-oscillatory near shocks or other types of discontinuities. The key idea to ensure approximate solutions without spurious oscillations is to use a nonlinear dissipation mechanism that adds stabilizing numerical diffusion near discontinuities but not in the smooth parts of the solution. Herein, we will briefly review two classes of such methods, flux-limiter and slope-limiter schemes.

9.5.1 Flux-limiter schemes

In Section 9.3, we saw that using (9.9) with a low-order flux approximations, like the upwind or the Lax–Friedrichs flux, gives low resolution in smooth

parts of the solution and tends to smear discontinuities. This means that the resulting schemes are stable, but generally lack the resolution that prevents decay near extremal points in the solution. Using a higher-order flux approximation like the Lax–Wendroff method, on the other hand, tends to introduce nonphysical oscillations near discontinuities. Ideally, we would like to have a method that behaves like a low-order scheme near discontinuities and like a higher-order scheme on smooth parts of the solution. The resulting scheme would then have second or higher order formal accuracy in smooth regions of the solution and at the same time have the stability of a first-order scheme where the solution is not smooth.

If $F_L(u^n; i)$ denotes a low-order flux like the upwind or the Lax–Friedrichs flux and $F_H(u^n; i)$ is a high-order flux like the Lax–Wendroff flux, we could imagine a new flux function of the form,

$$F(u^n; i + 1/2) = (1 - \theta_i^n) F_L(u^n; i + 1/2) + \theta_i^n F_H(u^n; i + 1/2) \quad (9.30)$$

where θ_i^n is an ‘intelligent’ quantity that measures the smoothness of the solution at grid cell i at time t_n and is defined such that θ_i^n is close to unity if the solution is smooth and close to zero if the solution is discontinuous. To give this sort of behavior, $\theta_i^n = \theta(u^n; i)$ must be a nonlinear function of the cell averages, and is referred to as a *limiter*. Inserting (9.30) into the conservative scheme (9.9) gives what is commonly called a *flux-limiting method*. A large number of successful high-resolution schemes have been developed using this approach, but a more detailed discussion is beyond the scope of this book.

9.5.2 Slope-limiter schemes

As an alternative to flux limiting, we can use a more geometrical approach. When we developed the classical schemes in Section 9.3, we had to reconstruct a piecewise continuous approximation from the cell averages u_i^n to obtain the point values we needed evaluate the flux integral (9.10). The resulting fluxes could then be used to evolve the cell averages \bar{u}_i or u_i^n forward in time using (9.7) or (9.9), respectively. This approach can be generalized to the following three-step procedure:

1. Starting from known cell averages u_i^n , **reconstruct** a piecewise polynomial function $\hat{u}(x, t^n)$ defined for all x . We have already considered the simple choice of a piecewise constant approximation,

$$\hat{u}(x, t_n) = u_i^n, \quad x \in [x_{i-1/2}, x_{i+1/2}).$$

This will generally result in a method that is formally first order. To obtain a method of higher order, we use a piecewise polynomial interpolant $p_i(x)$ such that

$$\hat{u}(x, t_n) = \sum_i p_i(x) \chi_i(x).$$

Here, $\chi_i(x)$ is the characteristic function of the i th grid cell $[x_{i-1/2}, x_{i+1/2}]$. For a second-order scheme, we therefore use a piecewise bilinear reconstruction, in a third order scheme a piecewise biquadratic reconstruction, and so on.

2. We then **evolve** the differential equation, exactly or approximately, using $\hat{u}(x, t_n)$ as initial data.
3. Finally, we **average** the evolved solution $\hat{u}(x, t_{n+1})$ onto the grid again to obtain new cell averages u_i^{n+1} .

This three-step algorithm is often referred to as the *REA algorithm*, from the words reconstruct-evolve-average.

As an example of the reconstruction step, let us consider a linear reconstruction within each cell,

$$\hat{u}(x, t_n) = u_i^n + s_i(x - x_i), \quad x \in [x_{i-1/2}, x_{i+1/2}].$$

Here, the slope s_i can be estimated from the cell-average values of the neighboring cells. There are three obvious candidate stencils for estimating this slope,

$$s_i^- = \frac{u_i^n - u_{i-1}^n}{\Delta x}, \quad s_i^+ = \frac{u_{i+1}^n - u_i^n}{\Delta x}, \quad s_i^c = \frac{u_{i+1}^n - u_{i-1}^n}{2\Delta x}.$$

The two one-sided differences are first-order approximations for smooth data, whereas the central difference is second order and would generally be the preferred choice to maximize spatial accuracy. However, one can show that all three choices lead to schemes that are formally second-order accurate on smooth solution. Near discontinuities, on the other hand, care must be taken to avoid introducing spurious oscillations, and which of the three stencils we should choose will depend on the local behavior of the underlying function, or in our case, on how the values $u_{i-1}^n, u_i^n, u_{i+1}^n$ vary relative to each other. To illustrate this, let us consider a simple step function,

$$u_k^n = \begin{cases} 1, & k \leq i, \\ 0, & k > i. \end{cases}$$

In this case, $s_i^- = 0$, $s_i^+ = -1/\Delta x$, and $s_i^c = -1/2\Delta x$. Thus, $\hat{u}(x, t_n)$ will remain monotone if we choose $s_i = s_i^-$, whereas a new maximum is introduced if we either choose s_i^+ or s_i^c as illustrated in Figure 9.5.

As we saw in the previous section, the entropy-weak solution of the conservation law has diminishing total variation and it can also be shown that the solution is bounded in the L^∞ norm so that $\|u(\cdot, t)\|_\infty \leq \|u_0\|_\infty$. Introducing a new maximum will obviously violate both these properties. Therefore we need to put some “intelligence” into the stencil computing the linear slopes. The key to obtain this intelligence lies in the introduction of a nonlinear averaging function Φ capable of choosing the correct slope. Given Φ , we simply let

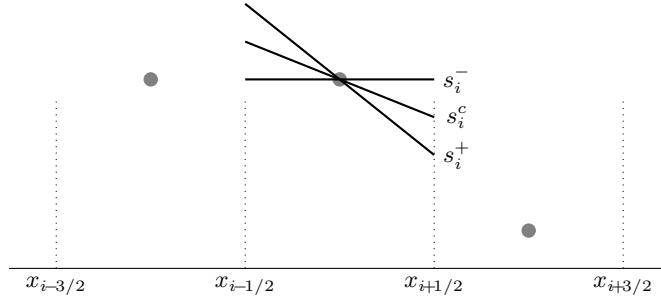


Fig. 9.5. Reconstruction of a piecewise linear function inside cell $[x_{i-1/2}, x_{i+1/2}]$ using three different candidate slopes. The dots signify the cell-averaged values u_{i-1}^n , u_i^n , and u_{i+1}^n .

$$\Delta x s_i = \Phi(u_i^n - u_{i-1}^n, u_{i+1}^n - u_i^n), \quad \Phi(a, b) = \phi\left(\frac{b}{a}\right)a.$$

This function is called a *limiter* and has much of the same purpose as the flux limiter θ_i^n . The limiter introduces a nonlinearity in the scheme and under certain restrictions on the function ϕ one can show that it satisfies the desired TVD property. Examples of limiters include the minmod, modified minmod, Sweby limiters

$$\phi^{MM}(r) = \max(0, \min(r, 1)), \quad (9.31)$$

$$\phi^{MM}(r; \theta) = \max(0, \min(\theta r, \frac{1+r}{2}, \theta)), \quad \theta \in [1, 2], \quad (9.32)$$

$$\phi^{Sw}(r; \theta) = \max(0, \min(\theta r, 1), \min(r, \theta)), \quad \theta \in [1, 2], \quad (9.33)$$

The minmod limiter is a robust choice in the sense that always chooses the one-sided slope with the least magnitude away from extrema, and picks a constant reconstruction ($\phi = 0$) if the one-sided slopes have different sign. The modified minmod and the Sweby limiters, on the other hand, tend to pick steeper slopes, thus adding less numerical diffusion into the scheme. The larger θ is chosen, the steeper the slopes will be.

Next, let us consider the averaging step and assume for simplicity that we have already computed a piecewise constant reconstruction $\hat{u}(x, t_n)$. Going back to the sliding average $\bar{u}(x, t)$ from (9.6), we can write its evolution equation as follows,

$$\begin{aligned} \bar{u}(x, t + \Delta t) &= \bar{u}(x, t) \\ &- \frac{1}{\Delta x} \int_t^{t+\Delta t} \left[f(u(x + \frac{1}{2}\Delta x, s)) - f(u(x - \frac{1}{2}\Delta x, s)) \right] ds. \end{aligned} \quad (9.34)$$

This will leave us with two choices that lead to two fundamentally different classes of methods. When the piecewise constant reconstruction $\hat{u}(x, t_n)$ is used as initial data to evolve the differential equation, we get an initial-value

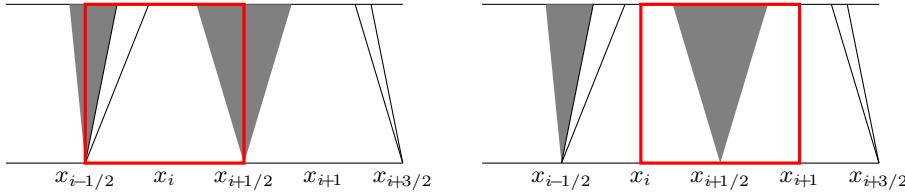


Fig. 9.6. Computation of the sliding average for upwind schemes (left) and centered schemes (right). The red lines show the integration volume in the (x, t) , while the shaded triangles and solid lines emanating from $x_{i\pm 1/2}$ illustrate the self-similar Riemann solutions resulting from using piecewise constant initial data.

problem consisting of a sequence of Riemann problems defined at each interface between neighboring grid cells. From our discussion of the front-tracking method in Section 8.4.3, we know that the solution to the initial-value problem for some small time will consist of the constant states u_i^n separated by the waves that make up the self-similar Riemann fans emanating from $x_{i-1/2}$, $x_{i+1/2}$, etc. This is illustrated in Figure 9.6. In *upwind* methods, we choose $x = x_i$ in (9.34) and integrate $f(u(\cdot, t))$ in time along the lines $(x_{i\pm 1/2}, t)$, which lie inside the self-similar Riemann fans $v_{i\pm 1/2}(x/t)$ defined by the states u_i^n and $u_{i\pm 1}^n$. Hence, we need to know the values $v_{i\pm 1/2}(0)$, but once we know these, we can integrate $f(u(\cdot, t))$ from time t to time $t + \Delta t$, where the maximum value of Δt is given by the CFL condition (9.20). In *centered* methods, we choose $x = x_{i+1/2}$ and thus compute the sliding average over a *staggered* grid cell $[x_i, x_{i+1}]$. The advantage of this is that the flux integral is evaluated at the points x_i and x_{i+1} , where the solution to the initial-value problem is given by the constants u_i^n and u_{i+1}^{n+1} , respectively. If we choose our discretization parameters so that Δt satisfies the following CFL condition,

$$\max_i |f'_c(\cdot; u_i^n, u_{i+1}^n)| \frac{\Delta t}{\Delta x_i} \leq \frac{1}{2}. \quad (9.35)$$

we are guaranteed that the solution to the initial-value problem will stay smooth at the points x_i, x_{i+1} . The flux integrals can thus be computed using some standard integration scheme, and the only information we need to know about the Riemann problems is an upper bound on how fast the waves are propagating. To this end, we could typically use $\max_u |f'(u)|$, which gives a somewhat less sharp estimate than (9.35).

We now have all the ingredients we need to develop specific high-resolution schemes. As an example, let us develop the second-order, non-oscillatory centered scheme proposed by Nessyahu and Tadmor [191], which is an example of a *staggered scheme*, in which the grid cells are shifted half a cell distance back and forth every time step. For simplicity, we assume that the grid has uniform cell size Δx . Let u_i^n approximate the cell-average over the i th cell $[x_{i-1/2}, x_{i+1/2}]$ at time $t_n = n\Delta t$ and $u_{i+1/2}^{n+1}$ the cell-average over the staggered cell $[x_i, x_{i+1}]$ at time $t_{n+1} = (n+1)\Delta t$. Since we seek a second-order

method, the scheme starts with a piecewise linear reconstruction,

$$\hat{u}_i(x, t_n) = u_i^n + (x - x_i)s_i, \quad x \in [x_{i-1/2}, x_{i+1/2}].$$

Inserting this into reconstruction into (9.34) and evaluating the sliding average over the staggered grid cell $[x_i, x_{i+1}]$ gives

$$\begin{aligned} u_{i+1/2}^{n+1} &= \frac{1}{\Delta x} \int_{x_i}^{x_{i+1}} \hat{u}(x, t) dx - \frac{1}{\Delta x} \int_{t_n}^{t_{n+1}} [f(\hat{u}(x_{i+1}, t)) - f(\hat{u}(x_i, t))] dt \\ &= \frac{1}{2}(u_i^n + u_{i+1}^n) + \frac{\Delta x}{8}(s_i - s_{i+1}) \\ &\quad - \frac{1}{\Delta x} \int_{t_n}^{t_{n+1}} [f(\hat{u}(x_{i+1}, t)) - f(\hat{u}(x_i, t))] dt. \end{aligned}$$

To get second-order accuracy, we use the midpoint rule to approximate the flux integrals,

$$\int_{t_n}^{t_{n+1}} f(\hat{u}(x_i, t)) dt \approx \Delta t f(\hat{u}(x_i, t_n + \frac{1}{2}\Delta t)).$$

Looking back at Figure 9.6, we see that the point values $\hat{u}(x_i, t_{n+1/2})$ and $\hat{u}(x_{i+1}, t_{n+1/2})$ lie in the region where the solution is smooth as long as the time step Δt satisfies a CFL condition of $\frac{1}{2}$. We can therefore use a Taylor series, in which we insert the partial differential equation, to extrapolate the solution forward in time from t_n ,

$$\begin{aligned} \hat{u}(x_i, t_{n+1/2}) &\approx \hat{u}(x_i, t_n) + \frac{\Delta t}{2} \partial_t \hat{u}(x_i, t_n) \\ &= \hat{u}(x_i, t_n) - \frac{\Delta t}{2} \partial_x f(\hat{u}(x_i, t_n)) \approx u_i^n - \frac{\Delta t}{2} \sigma_i. \end{aligned}$$

The flux gradient σ_i can either be compute directly as $f'(u_i^n)s_i$ since the point value $\hat{u}(x_i, t_n)$ coincides with the cell average u_i^n for a linear reconstruction, or we can reconstruct the slope from a piecewise linear reconstruction of the fluxes of the cell averages. Summarizing, we have developed the following predictor-corrector scheme,

$$\begin{aligned} s_i &= \Phi(u_i^n - u_{i-1}^n, u_{i+1}^n - u_i^n), \\ \sigma_i &= \Phi(f(u_i^n) - f(u_{i-1}^n), f(u_{i+1}^n) - f(u_i^n)), \\ u_i^{n+1/2} &= u_i^n - \frac{r}{2} \sigma_i, \\ g_i^n &= f(u_i^{n+1/2}) + \frac{1}{8r} s_i, \\ u_{i+1/2}^{n+1} &= \frac{1}{2}(u_i^n + u_{i+1}^n) - r(g_{i+1}^n - g_i^n). \end{aligned} \tag{9.36}$$

This scheme is formally second-order accurate and will, unlike the classical schemes introduced in Section 9.3, produce approximate solutions that are bounded by their initial data in L^∞ and have diminishing total variations. See `nt.m` in the `book` module for a simple implementation.

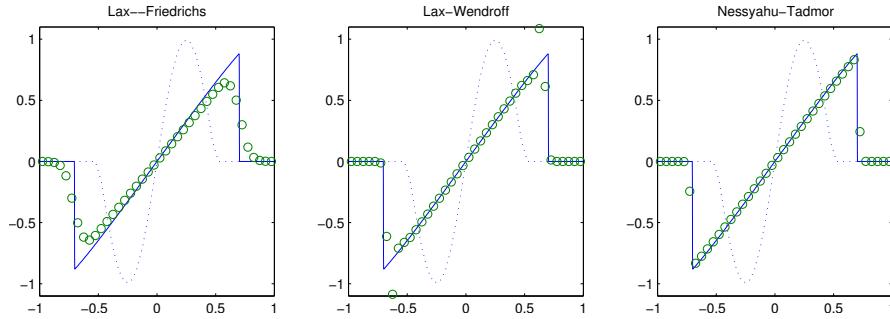


Fig. 9.7. Approximate solutions of Burgers' equation at time $t = 0.6$ computed with two classical and a high-resolution scheme. Dashed lines represent the initial data.

Example 9.4. To demonstrate the difference between classical and high-resolution schemes, we can study Burgers' equation,

$$u_t + \left(\frac{1}{2}u^2\right)_x = 0, \quad u(x, 0) = u_0(x).$$

The equation has a convex flux and is often used as a conceptual model for the momentum equation in more comprehensive fluid models. As a simple test case, we set $u_0(x, t) = \sin(2\pi x)\chi_{[-1/2, 1/2]}(x)$. Figure 9.7 shows approximate solutions at time $t = 0.6$ computed by the Lax-Friedrichs, the Lax-Wendroff, and the Nessyahu-Tadmor scheme. Burgers' equation has positive characteristics for $u > 0$ and negative characteristics for $u < 0$. The positive part of the smooth initial profile will therefore start to propagate to the right, steepen, and form a decaying shock wave that gradually propagates slower and slower. The negative part of the smooth wave will show a similar leftward propagation. As expected, the Lax-Friedrichs scheme smears the shock waves, whereas the Lax-Wendroff scheme introduces oscillations behind the shocks. For the Nessyahu-Tadmor scheme we have used the modified minmod limiter (9.32) with $\theta = 2$, which ensures that the leading shocks are resolved within two grid cells in a non-oscillatory manner.

9.5.3 Semi-discrete schemes

As an alternative to fully-discrete schemes, one can also use (9.7) to develop a semi-discrete scheme. In my opinion, the semi-discrete framework is more elegant, is simpler to work with in multiple spatial dimensions, and makes it simpler to include second-order terms. The disadvantage is that semi-discrete schemes are not always as accurate as their fully-discrete counterparts. In the following, we will describe a general semi-discrete framework for 2D conservation laws. For simplicity, we assume that $\Delta x = \Delta y$ and let u_{ij} denote the extension of (9.6) to a two-dimensional grid cell $[x_{i-1/2}, x_{i+1/2}] \times [y_{j-1/2}, y_{j+1/2}]$. Then, the evolution equation for u_{ij} reads,

$$\begin{aligned}\frac{d}{dt}u_{ij}(t) = L_{ij}(u(t)) = & -\frac{1}{\Delta x}\left[F_{i+1/2,j}(t) - F_{i-1/2,j}(t)\right] \\ & -\frac{1}{\Delta y}\left[G_{i,j+1/2}(t) - G_{i,j-1/2}(t)\right],\end{aligned}\quad (9.37)$$

where $F_{i\pm 1/2,j}$ and $G_{i,j\pm 1/2}$ are numerical approximations to the fluxes over the cell edges

$$\begin{aligned}F_{i\pm 1/2,j}(t) &\approx \frac{1}{\Delta y} \int_{y_{j-1/2}}^{y_{j+1/2}} f(u(x_{i\pm 1/2}, y, t)) dy, \\ G_{i,j\pm 1/2}(t) &\approx \frac{1}{\Delta x} \int_{x_{i-1/2}}^{x_{i+1/2}} g(u(x, y_{j\pm 1/2}, t)) dx.\end{aligned}\quad (9.38)$$

To get a specific scheme, we need to choose the following ingredients: quadrature rules to approximate the flux integrals (9.38); a piecewise polynomial reconstruction scheme to compute approximate point values of u at the integration points; a numerical flux function to compute the correct flux across the interface based on the two one-sided point values on opposite sides of the interface; and a numerical scheme to integrate the ODEs (9.37).

Let us start with the quadrature rule. To get second-order accuracy, it is sufficient to use the midpoint rule so that $F_{i\pm 1/2,j}$ is evaluated in the points $(x_{i\pm 1/2}, y_j)$ and $G_{i,j\pm 1/2}$ in the points $(x_i, y_{j\pm 1/2})$. To get third order spatial accuracy, some authors prefer to use Simpson's rule, but my favorite is to use the fourth-order open Gauss quadrature with points $(x_{i\pm 1/2}, y_j \pm \Delta y / 2\sqrt{3})$ and $(x_i \pm \Delta x / 2\sqrt{3}, y_{j\pm 1/2})$. If we let $x_{i\pm\alpha}$ and $y_{j\pm\alpha}$ denote the Gauss integration points, the edge fluxes (9.38) become,

$$\begin{aligned}F_{i\pm 1/2,j}(t) &= \frac{1}{2} \left[f(u(x_{i\pm 1/2}, y_{j+\alpha}, t)) + f(u(x_{i\pm 1/2}, y_{j-\alpha}, t)) \right], \\ G_{i,j\pm 1/2}(t) &= \frac{1}{2} \left[g(u(x_{i+\alpha}, y_{j\pm 1/2}, t)) + g(u(x_{i-\alpha}, y_{j\pm 1/2}, t)) \right].\end{aligned}\quad (9.39)$$

To get point values at the integration points, we need to apply a piecewise polynomial reconstruction. The simplest possible approach is to apply the linear construction we discussed above independently in each direction,

$$\hat{u}_{ij}(x, y, t^n) = u_{ij}^n + s_{ij}^x(x - x_i) + s_{ij}^y(y - y_j), \quad (9.40)$$

which will give us a piecewise bilinear reconstruction and lead to schemes of second-order spatial accuracy. Higher spatial order can be obtained by using a so-called essentially non-oscillatory (ENO) or a weighted essentially non-oscillatory (WENO) reconstruction; see [234] for more details.

The piecewise polynomial reconstruction will give one-sided point values that are generally different on opposite sides of the grid interface. To be able to compute (9.39), we need to specify how these one-sided point values should be used to evaluate the integrand of (9.38). We have already encountered a number of different fluxes, including the upwind flux which only works

when the flux function only has positive or negative derivatives, the upstream mobility-weighted flux (9.21) which only applies to flow in porous media, and the general Lax–Friedrichs (9.13) and Lax–Wendroff (9.16) fluxes. My personal favorite, however, is the central-upwind flux [143],

$$F^{CUW}(u_L, u_R) = \frac{a^+ f(u_L) - a^- f(u_R)}{a^+ - a^-} + \frac{a^+ a^-}{a^+ - a^-} (u_R - u_L). \quad (9.41)$$

Here, the values (a^-, a^+) are estimates of how far the Riemann fan emanating from the discontinuity (u_L, u_R) extends in the negative and positive direction. Obtaining such estimates is typically much simpler than solving the Riemann problem. Using information of how fast the Riemann fan propagates in each direction, the evolving solution can be divided into two unions of local domains: one covering the local Riemann fans, where the solution is possibly discontinuous and one covering the area in-between. The specific (9.41) follows if we evolve and average the solution separately in these two sets of regions. The central-upwind flux has a number of interesting special cases. For monotone flux functions, the flux reduces to the standard upwind method. For example, for scalar equations with $f'(u) \geq 0$, a^- is zero and $F^{CUW} = f(u_L)$. If we use a constant reconstruction so that $u_L = u_i$ and $u_R = u_{i+1}$, we get a semi-discrete version of the famous Harten–Lax–van Leer (HLL) upwind scheme [100]. Finally, if $a^+ = -a^- = \Delta x / \Delta t$, (9.41) reduces to the dissipative Lax–Friedrichs flux.

We now have all we need to evaluate the flux integrals (9.38). The remaining ingredient is then to choose a method to integrate the ordinary differential equations (9.37) in time. Using the standard forward Euler method would give a scheme that is first-order accurate in time and to retain high-order accuracy also in time it is common to use an explicit Runge–Kutta predictor-corrector scheme. Among the many different types of such methods, the best choice is to use so-called TVD Runge–Kutta methods [233], which have been developed especially to integrate semi-discrete schemes in time without creating spurious oscillations. The second-order version of the method reads

$$\begin{aligned} u_{ij}^{(1)} &= u_{ij}^n + \Delta t L_{ij}(u^n), \\ u_{ij}^{n+1} &= \frac{1}{2} u_{ij}^n + \frac{1}{2} [u_{ij}^{(1)} + \Delta t L_{ij}(u^{(1)})]. \end{aligned} \quad (9.42)$$

The function `cuw` in the `book` module implements one such semi-discrete scheme in 1D with piecewise linear reconstruction, central-upwind flux (9.41), and second-order time integration (9.42).

Example 9.5. Let us revisit the test cases from Example 9.1 on page 287 and Example 9.2 on page 290. Figure 9.8 shows solutions computed by the fully-discrete `nt` and the semi-discrete `cuw` solvers. While the fully-discrete scheme gives excellent resolution of both the smooth and the discontinuous parts of the advecting waves, the semi-discrete scheme has smeared the profile to the point where it is not possible to tell which parts of the solution are continuous

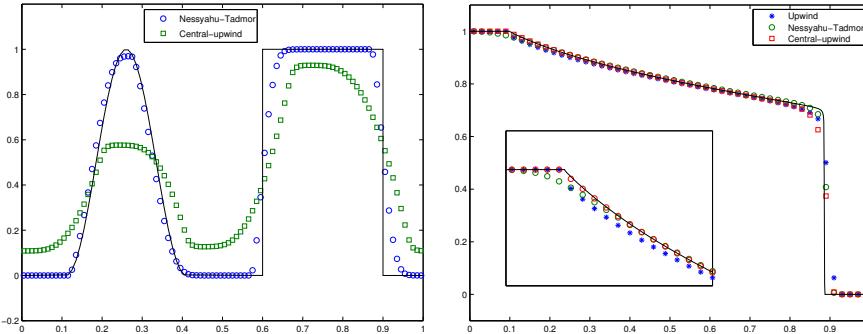


Fig. 9.8. Approximate solutions for the linear advection problem with periodic boundary conditions at time $t = 20$ (left) and for the Buckley–Leverett problem at time $t = 0.65$ (right) computed by the fully-discrete Nessyahu–Tadmor scheme and the semi-discrete central-upwind scheme, both run with CFL number 0.495.

and which are discontinuous. The explanation is as follows: for a linear flux function $f(u) = u$ the central-upwind flux (9.41) reduces to the upwind flux. With this flux function, the advective profile will be exactly reproduced if the upwind scheme is run with a CFL number equal unity; this explains the excellent resolution observed for the upwind scheme using a CFL number of 0.995 in Figure 9.1. The central-upwind method, on the other hand, the stability restriction requires a CFL number of $1/2$ or less. A more fair comparison is therefore the simulation reported in Figure 9.2, where the upwind scheme was run with CFL number $1/2$. Comparing these two solutions, we see that the first-order scheme gives significantly more smearing than the high-resolution scheme.

For the Buckley–Leverett profile, the main difference between the upwind scheme and the two high-resolution schemes is how they resolve the rarefaction wave and the kink. Here, the semi-discrete scheme appears to be the most accurate.

9.6 Discretization on unstructured polyhedral grids

So far, this chapter has given you a brief introduction to various types of numerical methods that can be used to discretize saturation equations. The methods are all reasonably simple to implement on regular Cartesian and rectilinear grids and on grids consisting entirely of simplices (triangles in 2D and tetrahedrons in 3D). However, as you have seen multiple times throughout this book, realistic grid models are seldom that simple and for the baseline solvers in MRST we have therefore chosen to go with the most robust choice of them all, namely the explicit or implicit version of the single-point upstream mobility weighting scheme. In this section we will therefore discuss in more

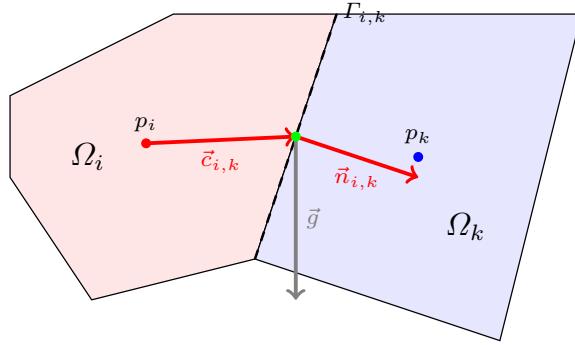


Fig. 9.9. Two cells used to define the finite-volume discretization of the general two-phase transport equation (9.43). Here, the interface $\Gamma_{i,k}$ between the two cells Ω_i and Ω_k has normal vector $\vec{n}_{i,k}$ and area $A_{i,k}$.

detail how to formulate this scheme on unstructured, polyhedral grids for a general transport equation

$$\phi \frac{\partial S}{\partial t} + \nabla \cdot \vec{H}(S) = 0, \quad (9.43)$$

where the flux function \vec{H} includes viscous flow driven by pressure gradients as well as gravity segregation and capillary forces,

$$\begin{aligned} \vec{H}(S) &= \frac{\lambda_w}{\lambda_w + \lambda_n} \vec{v} + \frac{\lambda_w \lambda_n K}{\lambda_w + \lambda_n} (\Delta \rho \vec{g} + \nabla P_c(S)) \\ &= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S). \end{aligned} \quad (9.44)$$

To discretize this equation, we start from a multi-dimensional analogue of (9.8),

$$S_i^{n+1} - S_i^n = \frac{1}{\phi_i |\Omega_i|} \sum_k \int_{t_n}^{t_{n+1}} \int_{\Gamma_{ik}} \vec{H}(S(\vec{x}, t)) \cdot \vec{n}_{i,k} ds dt. \quad (9.45)$$

defined over a general grid cell Ω_i as illustrated in Figure 9.9. Since we do not want to develop a higher-order discretization, it is sufficient to evaluate the time integral at one of its end-point, so that (9.45) simplifies to

$$S_i^{n+1} - S_i^n = \frac{\Delta t}{\phi_i |\Omega_i|} \int_{\Gamma_{ik}} \vec{H}(S(\vec{x}, t_m)) \cdot \vec{n}_{i,k} ds, \quad m = n, n+1. \quad (9.46)$$

Before we describe the discretization of the three different transport terms H_f , H_g , and H_c , let us quickly recap how we discretize the viscous flux $\vec{v} = -K \nabla p$ in the corresponding pressure equation. Referring to Figure 4.10 on page 137, the flux reads

$$\begin{aligned} v_{i,k} &\approx A_{i,k} \mathsf{K}_i \frac{(p_i - \pi_{i,k}) \vec{c}_{i,k}}{|\vec{c}_{i,k}|^2} \cdot \vec{n}_{i,k} = T_{i,k}(p_i - \pi_{i,k}) \\ &= [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}(p_i - p_k), \end{aligned}$$

where the first line gives the one-sided formulation compute entirely from quantities associated with cell i , and the second line gives that formulation that also couples to the neighbouring cell k . From this expression, it follows naturally that the capillary term should be discretized as follows

$$A_{i,k} \nabla P_c(S) \cdot \vec{n}_{i,k} \approx [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}(P_c(S_i) - P_c(S_k)) = P_{i,k}(S). \quad (9.47)$$

Likewise, we can define a “gravity flux” g_{ik} that is independent of saturation,

$$g_{ik} = [g_{i,k}^{-1} + g_{k,i}^{-1}]^{-1}, \quad \begin{aligned} g_{i,k} &= (\Delta\rho)|_{\Omega_i} \mathsf{K}_i \vec{g} \cdot \vec{n}_{i,k}, \\ g_{k,i} &= (\Delta\rho)|_{\Omega_k} \mathsf{K}_k \vec{g} \cdot \vec{n}_{k,i}. \end{aligned} \quad (9.48)$$

With this, all we need to do is to find the correct upstream value λ_α^u for each of the two phases, and then the overall approximation of the interface flux will read as follows

$$H_{ik} = \frac{\lambda_w^u}{\lambda_w^u + \lambda_n^u} v_{ik} + \frac{\lambda_w^u \lambda_n^u}{\lambda_w^u + \lambda_n^u} [g_{ik} + P_{ik}]. \quad (9.49)$$

To determine the upstream directions, we need to compare the Darcy flux v_{ik} and the gravity flux g_{ik} . (If there are capillary forces, these are henceforth assumed to be added to the gravity flux, i.e., $g_{ik} + P_{ik} \rightarrow g_{ik}$). If v_{ik} and g_{ik} have the same sign, we know that the correct choice of the wetting mobility is from the left (right) of the interface if the fluxes are positive (negative). Likewise, when the two fluxes have different sign, we know that the correct choice of upstream value for the non-wetting fluid is independent of the actual mobility values. These cases can be summarized in the following tabular:

$\text{sign}(v_{ik})$	$\text{sign}(g_{ik})$	λ_w^u	$\text{sign}(v_{ik})$	$\text{sign}(g_{ik})$	λ_n^u
≥ 0	≥ 0	$\lambda_w(S_L)$	≥ 0	≤ 0	$\lambda_n(S_L)$
≤ 0	≤ 0	$\lambda_w(S_R)$	≤ 0	≥ 0	$\lambda_n(S_R)$

If neither of these cases are fulfilled we need to check the sign of the phase fluxes to determine the correct upstream direction. In practices, we do not check the phase fluxes but rather the phase fluxes divided by the fractional flow fuctions since the latter are less expensive to compute. That is, we check the sign of $v_{ik} + g_{ik} \lambda_n$ and $v_{ik} - g_{ik} \lambda_w$ and use this to pick the correct upstream value.

For most cases in reservoir simulation, we will also have source terms that drive flow, so that (9.43) is replaced by an inhomogeneous equation

$$\phi \frac{\partial S}{\partial t} + \nabla \cdot \vec{H}(S) = \max(q, 0) + \min(q, 0)f(S).$$

In the next chapter, we will describe how the resulting transport solvers have been implemented in MRST and discuss how to combine them pressure solvers in a sequential solutio procedure to solve incompressible, multiphase flow.

10

Solvers for Incompressible Immiscible Flow

The previous chapter introduced you to the basic equations describing multiphase flow in porous media. Multiphase flow is generally governed by a number of physical effects. For simple flows, fluids can be moved by pressure differentials, by gravity segregation, capillary forces, or fluid and rock compressibility. More advanced models also involve dissolution effects, hysteresis, and molecular diffusion, to name a few. Multiphase flow equations can therefore give rise to very different behavior depending upon what are the main physical effects. During the formation of petroleum reservoirs, the dominant effects are buoyancy and capillary forces, which govern how hydrocarbons migrate upward and enter new layers of consolidated sediments. The same effects are also thought to dominate the long-term behavior in geological carbon storage, where the buoyant CO₂ phase will tend to migrate upward in the formation long after its injection has ceased. In the recovery of petroleum resources, on the other hand, the predominant force is viscous advection caused by pressure differential. Here, pressure disturbances will in most cases propagate much faster through the porous medium than the material waves that transport fluid phases and chemical components. This means that the mathematical equations describing multiphase flow will have a mixed character: Whereas the pressure part of the problem has a relatively strong elliptic nature, the material waves will have a more hyperbolic character. This mixed nature is one of the main reasons why solving multiphase flow equations turns out to be relatively complicated. In addition, we have all the other difficulties that we already have encountered for single-phase flow in Chapters 5 to 7. The variable coefficients entering the flow equations typically are highly heterogeneous with orders of magnitude variation and complex correlations involving a wide range of spatial correlation lengths. The grids used to describe real geological media tend to be highly complex, having unstructured topologies, irregular cell geometries, and orders of magnitude aspect ratios. The dominant flow in injection and production wells takes place on a small scale relative to the reservoir and hence needs to be modeled using approximate analytical expressions, and so on.

This chapter will teach you how to discretize and solve the multiphase flow equations in the special case of incompressible rock and immiscible and incompressible fluids. The system of PDEs can then be reformulated so that it consists of an elliptic equation for fluid pressure and one or more transport equations. These transport equations are generally parabolic, but have a strong hyperbolic character (see Section 8.3). Since the pressure and saturations equations have very different mathematical characteristics, it is natural to solve them in consecutive substeps. Sequential solution procedures and incompressible flow models are popular in academia and for research purposes, but are less used in industry. To the extent simulations are used for practical reservoir engineering, they are mainly based on compressible equations and solution procedures in which flow and transport are solved as a coupled system. Such approaches are very robust and particularly useful for problems with large variations in time constants or strong coupling between different types of flow mechanisms. We will return to compressible three-phase flow in Chapter 11. In the following, we only consider two phases.

10.1 Fluid objects for multiphase flow

In Chapter 5, we discussed the basic data objects that enter a flow simulation. When going from a single-phase to a multiphase flow model, the most prominent changes take place in the fluid model. It is this model that generally will tell your solver how many phases are present and how these phases affect each other when flowing together in the same porous medium. We therefore start by briefly outlining a few fluid objects that implement the basic fluid behavior discussed in Chapter 8.

To describe an incompressible flow model, we need to know the viscosity and the constant density of each fluid phase, as well as the relative permeabilities of the fluid phases. If the fluid model includes capillary forces, we also need one or more functions that specify the capillary pressure as function of saturation. The most basic multiphase fluid model in MRST is the following,

```
fluid = initSimpleFluid('mu' , [ 1, 10]*centi*poise , ...
                      'rho' , [1014, 859]*kilogram/meter^3, ...
                      'n' , [ 2, 2]);
```

which implements a simplified version of the Corey model (8.15) in which the residual saturations S_{wr} and S_{nr} are assumed to be zero and the end-points are scaled to unity, so that $k_{rw} = (S_w)^{n_w}$ and $S_{rn} = (1 - S_w)^{n_n}$. To recap from Chapter 5, the fluid object offers the following interface to evaluate the petrophysical properties of the fluid:

```
mu = fluid.properties();      % gives mu_w and mu_n
[mu,rho] = fluid.properties(); % .... plus rho_w and rho_n
```

New to multiphase flow is the `relperm` function, which takes a single fluid saturation or an array of fluid saturations as input and outputs the corresponding values of the relative permeabilities. Plotting the relative permeability curves of the `fluid` object can be achieved by the following code

```
s=linspace(0,1,20)';
kr = fluid.relperm(s);
plot(s,kr(:,1),'-s',s,kr(:,2),'-o');
```

The `relperm` function can also return the first and second derivatives of the relative permeability curves when called with two or three output arguments.

The basic fluid model does not have any capillary pressure. To also include this effect, one can use another fluid model,

```
fluid = initSimpleFluidPc('pc_scale', 2*barsa);
```

which adds a capillary function that assumes a linear relationship $P_c(S) = C(1 - S)$. The resulting fluid object has an additional function pointer that can be used to evaluate capillary pressure

```
fluid =
properties: @(varargin) properties(opt,varargin{:})
saturation: @(x,varargin)x.s
relperm: @(s,varargin) relperm(s,opt,varargin{:})
pc: @(state) pc_funct(state,opt)
```

The capillary pressure function `pc` is evaluated using a `state` object and not a saturation. This simple function can be extended to include Leverett J-function scaling (8.9) on page 242 by using the fluid object generated by `initSimpleFluidJfunc`.

The `incomp` module also implements the general Corey model with end-point scaling k_α^0 and nonzero residual saturations S_{wr} and S_{nr} .

```
fluid = initCoreyFluid('mu' , [ 1, 10]*centi*poise , ...
'rho' , [1014, 859]*kilogram/meter^3, ...
'n' , [ 3, 2.5] , ...
'sr' , [ 0.2, .15] , ...
'kwm' , [ 1, .85]);
```

Figure 10.1 shows the relative permeabilities and their first and second derivatives for this particular model.

COMPUTER EXERCISES:

58. Modify the Corey model so that it also can output the residual saturations and the end-point scaling values
59. Implement the Brooks–Corey (8.16) and the van Genuchten models (8.17) and (8.18)
60. Extend the models to also include the capillary functions (8.11) and (8.12)

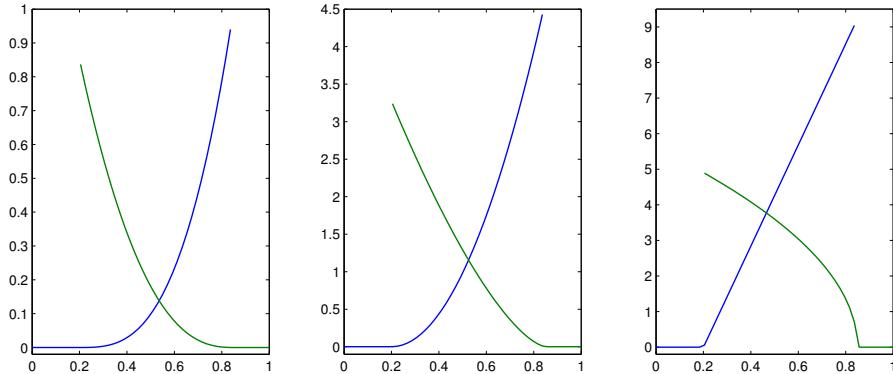


Fig. 10.1. Corey relative permeabilities (left) and their first and second derivatives (middle and right) constructed by the `initCoreyFluid` function.

10.2 Sequential solution procedures

To solve the two-phase, incompressible model we rely entirely on the fractional-flow formulation developed in Section 8.3.2 on page 253. As you may recall, this model consists of an elliptic pressure equation

$$\nabla \cdot \vec{v} = q, \quad \vec{v} = -\boldsymbol{\lambda}(\nabla p_n - f_w \nabla P_c - (\rho_w f_w + \rho_n f_n) g \nabla z) \quad (10.1)$$

and a parabolic transport equation

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot [f_w(\vec{v} + \boldsymbol{\lambda}_n(\Delta \rho g \nabla z + \nabla P_c))] = q_w. \quad (10.2)$$

Here, the capillary pressure $p_c = p_w - p_n$ is assumed to be a known function P_c of the wetting saturation S_w , and the transport equation becomes hyperbolic whenever P'_c is zero.

In the standard sequential solution procedure, the system (10.1)–(10.2) is evolved in time using a set of discrete time steps Δt_i . Let us assume that p , \vec{v} , and S_w are all known at time t and that we want to evolve the solution to time $t + \Delta t$. At the beginning of the time step, we first assume that the saturation S_w is fixed. This means that the parameters $\boldsymbol{\lambda}$, f_w , and f_n in (10.1) become functions of the spatial variable \vec{x} , and hence we can use this equation only to update the pressure p_n and the Darcy velocity \vec{v} . Next, \vec{v} and p_n are held fixed while (10.2) is evolved a time step Δt to define an updated saturation $S_w(\vec{x}, t + \Delta t)$. This saturation is then held fixed when we update p_n and \vec{v} in the next time step, and so on.

Some authors refer to this solution procedure as an *operator splitting* method since the solution procedure effectively splits the overall solution operator of the flow model into two parts that are evolved in consecutive substeps. Likewise, some authors refer to the sequential solution procedure as IMPES,

which is short-hand for *implicit pressure, explicit saturation*. Using the name IMPES is strictly speaking only correct if the saturation evolution is approximated by a single time step of an explicit transport solver. The size of the splitting step Δt is then restricted by the CFL condition of the explicit scheme. In many cases, the overall flow system does not have stability requirements that necessitate such a restriction on Δt . Indeed, by writing the flow model in the fractional-flow formulation, we have isolated the parts of the system that have stability restrictions to the hyperbolic saturation equation. The elliptic pressure equation, on the other hand, describes smooth solutions resulting from the instant redistribution of pressure in a system with infinite speed of propagation. For this equation, we can therefore in principle use as large time step as we want. In other words, as long as we ensure that the evolving discontinuities and sharp transitions are propagated in a stable manner in the saturation equation, our only concern when choosing the size of the splitting step should be to control or minimize the *splitting error* introduced by accounting for pressure and transport in separate substeps. As pointed out in Section 8.3.2, the fractional flow formulation underlying our operator splitting has been developed so that it minimizes the coupling between saturation and pressure. For incompressible flow models, the effect dynamic changes in the saturation field have on the pressure is governed entirely by the total mobility $\lambda(S)$, which in many cases is a function that locally has small and relatively smooth variation in time. For this reason, one can typically use splitting steps that are significantly larger than the CFL restriction of the hyperbolic part of the saturation equation and still accurately resolve the coupling between pressure and saturation. In other words, for each pressure update, the saturation can be updated by an explicit solver using multiple saturation substeps, or by an implicit solver using either a single or multiple saturation substeps. If necessary, one can also iterate on the splitting steps.

10.2.1 Pressure solvers

The pressure equation (10.1) for incompressible, multiphase flow is time dependent. This time dependence comes as the result of three factors:

- K/μ being replaced by the total mobility $\lambda(S_w)$, which depends on time through the saturation $S_w(\vec{x}, t)$,
- the constant density ρ being replaced by a saturation-dependent quantity $\rho_w f_w(S_w) + \rho_n f_n(S_w)$, and
- the source term q being replaced by a saturation-dependent source term $q - \nabla \lambda_w(S_w) \nabla P_c(S_w)$.

However, once S_w is held fixed in time, all three quantities become functions of \vec{x} only, and we hence end up again with an elliptic Poisson-type equation having the same spatial variation as in (4.10) on page 123. Hence, we can either use the two-point scheme introduced in Section 4.4.1 or one of the consistent discretization methods from Chapter 6, *mutatis mutandis*. The `incompTPFA` solver discussed in Chapter 5 and the `incompMimetic` and `incompMPFA` solvers

from Chapter 6 have been implemented so that they solve the pressure equation for a general system of m incompressible phases. Whether this system has one or more phases is determined by the fluid object and the reservoir state introduced in Section 5.1.2. We will therefore not discuss the pressure solvers in more detail.

10.2.2 Saturation solvers

Apart from the time loop, which we have already encountered in Chapter 7, the only remaining part we need is a solver for the transport equation (10.2) that implements the discretizations we introduced in Section 9.6. Summarized, this can be written on the following residual form for each cell Ω_i

$$\mathcal{F}_i(s, r) = s_i - r_i + \frac{\Delta t}{\phi_i |\Omega_i|} \left[\sum_k H_{ik}(s) - \max(q_i, 0) - \min(q_i, 0) f(S_i) \right], \quad (10.3)$$

where subscript s and r are cell-averaged quantities and subscript i refers to the cell the average is evaluated in. The sum of the interface fluxes for cell i

$$H_i(s) = \sum_k \frac{\lambda_w^u(s_i, s_k)}{\lambda_w^u(s_i, s_k) + \lambda_n^u(s_i, s_k)} [v_{ik} + \lambda_n^u(s_i, s_k)(g_{ik} + P_{ik})]. \quad (10.4)$$

is computed using the single-point, upstream mobility-weighting scheme discussed on page 308, whereas the fractional flow function f in the source term is evaluated from the cell average of S in cell Ω_i . The explicit scheme is given as $S^{n+1} = S^n - \mathcal{F}(S^n, S^n)$ and the implicit scheme follows as a coupled system of discrete nonlinear equations if we set $\mathcal{F}(S^{n+1}, S^n) = 0$. In the following, we will discuss the resulting solvers in a bit more detail.

Explicit solver

The `incomp` module offers the following explicit transport solver

```
state = explicitTransport(state, G, tf, rock, fluid, 'mech1', obj1, ...)
```

which evolves the saturation given in the `state` object a step `tf` forward in time. The function requires a complete and compatible model description consisting of a grid structure `G`, petrophysical properties `rock`, and a fluid model `fluid`. For the solver to be functional, the `state` object must contain the correct number of saturations per cell and an *incompressible* flux field that is consistent with the global drive mechanisms given by the `mech` argument ('`src`', '`bc`', and/or '`wells`') using correctly specified objects `obj`, as discussed in Sections 5.1.3–5.1.5. In practice, this means that the *input value* of `state` must be the *output value* of a previous call to an incompressible solver like `incompTPFA`, `incompMPFA`, or `incompMimetic`. In addition, the function takes a number of optional parameters that determine whether the time steps

are prescribed by the user or to be automatically computed by the solver. The solver can also ignore the Darcy flux and work as a pure gravity segregation solver if the optional parameter `onlygrav` is set to true. Finally, the solver will issue a warning if the updated saturation value is more than `satwarn` outside the interval [0, 1] of physically meaningful values (default value: `sqrt(eps)`).

The implicit and the explicit solvers involve many of the same operations and formulas used for the spatial discretizations. To avoid duplication of code we have therefore introduced a private helper function

```
[F,Jac] = twophaseJacobian(G, state, rock, fluid, 'pn1', pv1, ...)
```

that implements the residual form (10.3) and its Jacobian matrix $J = d\mathcal{F}$ and returns these as two function handles `F` and `Jac`. Using this function, the key lines of the explicit saturation solver read,

```
F = twophaseJacobian(G, state, rock, fluid, 'wells', opt.wells, ...);
s = state.s(:,1);
t = 0;
while t < tf,
    dt = min(tf-t, getdt(state));
    s(:) = s - F(state, state, dt);
    t = t + dt;
    s = correct_saturation(s, opt.satwarn);
    state.s = [s, 1-s];
end
```

Here, the function `getdt` implements a CFL restriction on the time step by estimating the maximum derivative of each function used to assemble interface fluxes and source terms (the interested reader can find details in the code). The function `correct_saturation` ensures that the computed saturations stay inside the interval of physically valid states. If this function issues a warning, it is highly likely that your time step exceeds the stability limit (or something is wrong with your fluxes or setup of the model).

Implicit solver

The implicit solver has the same user interface and parameter requirement as the explicit solver

```
state = implicitTransport(state, G, tf, rock, fluid, 'mech1', obj1, ...)
```

In addition, there are optional parameters that control the Newton–Raphson method used to solve for S^{n+1} . To describe this method, we start by writing the residual equations (10.3) for all cells in vector form

$$\mathbf{F}(\mathbf{s}) = \mathbf{s} - \mathbf{S} + \frac{\Delta t}{\phi|\Omega|} [\mathbf{H}(\mathbf{s}) - \mathbf{Q}^+ - \mathbf{Q}^- \mathbf{f}(\mathbf{s})] = \mathbf{0}. \quad (10.5)$$

Here, \mathbf{s} is the unknown state at time `tf` and \mathbf{S} is the known state at the start of the time step. As you may recall from Section 7.1, the Newton–Raphson

linearization of an equation like (10.5) can be written as,

$$\mathbf{0} = \mathbf{F}(\mathbf{s}_0 + \boldsymbol{\delta}\mathbf{s}) \approx \mathbf{F}(\mathbf{s}_0) + \nabla\mathbf{F}(\mathbf{s}_0)\boldsymbol{\delta}\mathbf{s},$$

which naturally suggests an iterative scheme in which the approximate solution $\mathbf{s}^{\ell+1}$ in the $(\ell + 1)$ -th iteration is obtained from

$$\mathbf{J}(\mathbf{s}^\ell)\boldsymbol{\delta}\mathbf{s}^{\ell+1} = -\mathbf{F}(\mathbf{s}^\ell), \quad \mathbf{s}^{\ell+1} \leftarrow \mathbf{s}^\ell + \boldsymbol{\delta}\mathbf{s}^{\ell+1}. \quad (10.6)$$

Here, $\boldsymbol{\delta}\mathbf{s}^{\ell+1}$ is referred to as the Newton update and \mathbf{J} is the Jacobian matrix. The `incomp` module was implemented before automatic differentiation was introduced in MRST and hence the Jacobian is computed analytically, using the following expansion

$$\begin{aligned} \mathbf{J}(\mathbf{s}) &= \frac{d\mathbf{F}}{d\mathbf{s}}(\mathbf{s}) = \mathbf{1} + \frac{\Delta t}{\phi|\Omega|} \left[\frac{d\mathbf{H}}{d\mathbf{s}}(\mathbf{s}) - \mathbf{Q}^{-} \frac{d\mathbf{f}}{d\mathbf{s}}(\mathbf{s}) \right], \\ \frac{d\mathbf{H}}{d\mathbf{s}} &= \frac{d\mathbf{H}}{d\boldsymbol{\lambda}_w} \frac{d\boldsymbol{\lambda}_w}{d\mathbf{s}} + \frac{d\mathbf{H}}{d\boldsymbol{\lambda}_n} \frac{d\boldsymbol{\lambda}_n}{d\mathbf{s}} + \mathbf{f}_w \boldsymbol{\lambda}_n \frac{d\mathbf{P}}{d\mathbf{s}}, \\ \frac{d\mathbf{H}}{d\boldsymbol{\lambda}_w} &= \frac{\mathbf{f}_w}{\boldsymbol{\lambda}} [\mathbf{v} + \boldsymbol{\lambda}_n(\mathbf{g} + \mathbf{P})], \quad \frac{d\mathbf{H}}{d\boldsymbol{\lambda}_n} = -\frac{\mathbf{f}_w}{\boldsymbol{\lambda}} [\mathbf{v} - \boldsymbol{\lambda}_w(\mathbf{g} + \mathbf{P})]. \end{aligned}$$

In general, we are not guaranteed that the resulting values in the vector $\mathbf{s}^{\ell+1}$ lie in the interval $[0, 1]$. To ensure physically meaningful saturation values, one can introduce a *line-search* method, which uses the Newton update to define a *search direction* $\mathbf{p}^\ell = \boldsymbol{\delta}\mathbf{s}^{\ell+1}$ and tries to find the value α that minimizes $h(\alpha) = F(\mathbf{s}^\ell + \alpha \mathbf{p}^\ell)$. One may either solve $h'(\alpha) = 0$ exactly, or use an *inexact line-search method* that only asks for a sufficient decrease in h . In the implicit solver discussed herein, we have chosen the latter approach and use an unsophisticated method in which α is reduced in a geometric sequence. The following code should give you the idea,

```
function [state, res, alph, fail] = linesearch(state, ds, target, F, ni)
    capSat = @(sat) min(max(0, sat),0);
    [alph,i,fail] = deal(0,0,true);
    sn = state;
    while fail && (i < ni),
        sn.s(:,1) = capSat(state.s(:,1) + pow2(ds, alph));
        res = F(sn);
        alph = alph - 1; i = i + 1;
        fail = ~norm(res, inf) < target;
    end
    alph = pow2(alph + 1); state.s = sn.s;
```

Here, F is a function handle to the residual function \mathbf{F} . The number of trials ni in the line-search method is set through the optional parameter '`lstrails`', while the target value is set as the parameter '`resred`' times the residual error upon entry. Default values for '`lstrails`' and '`resred`' are 20 and 0.99, respectively.

The implicit discretization is stable in the sense that there exists a solution S^{n+1} for an arbitrarily large time increment Δt . Unfortunately, there is no guarantee that we will be able to find this solution using the above line-search method. If the time step is too large, the Newton method may simply compute search directions that do not point us toward the correct solution. To compensate for this, we also need a mechanism that reduces the time step if the iteration does not converge and then uses a sequence of shorter time step to reach the prescribed time tf . First of all, we need to define what we mean by convergence; this is defined by the optional '`nltol`' parameter, which sets the absolute tolerance ϵ (default value 10^{-6}) on the residual $\|\mathcal{F}(S^{n+1}, S^n)\|_\infty \leq \epsilon$. In addition, we use a parameter '`maxnewt`' that gives the maximum number of iteration steps (default value 25) the method can take to reach a converged solution. The following code gives the essence of the overall algorithm of the iterative solver, as implemented in the helper function `newtonRaphson2ph`

```

mints = pow2(tf, -opt.tsref);
[t, dt] = deal(0.0, tf);
while t < tf && dt >= mints,
    dt = min(dt, tf - t);
    redo_newton = true;
    while redo_newton,
        sn_0 = resSol; sn = resSol; sn.s(:) = min(1,sn.s+0.05);
        res = F(sn, sn_0, dt);
        err = norm(res(:, inf));
        [nwtfail, linfail, it] = deal(err>opt.nltol, false, 0);
        while nwtfail && ~linfail && it < opt.maxnewt,
            J = Jac(sn, sn_0, dt);
            ds = -reshape(opt.LinSolve(J, reshape(res', [], 1)), ns, []);
            [sn, res, alph, linfail] = update(sn, sn_0, ds, dt, err);
            it = it + 1;
            err = norm(res(:, inf));
            nwtfail = err > opt.nltol;
        end
        if nwtfail,
            % Chop time step in two, or use previous successful dt
        else
            redo_newton = false;
            t = t + dt;
            % If five successful steps, increase dt by 50%
        end
    end
    resSol = sn;
end

```

Here, we have used two optional parameters: '`tsref`' with default value 12 gives the number of times we can halve the time step, while '`LinSolve`' is the linear solver, which defaults to `mldivide`. More details about the solver can be found by reading the code.

10.3 Simulation examples

You have now been introduced to all the functionality you need to solve incompressible, multiphase flow problems. It is therefore time to start looking into the qualitative behavior of such systems. In this section, we will discuss examples that highlight many of the typical multiphase phenomena you may encounter in practice. The examples are designed to highlight individual effects, or combinations of effects, and may not always be fully realistic in terms of physical scales, magnitude of the parameters and effects involved, etc. We will also briefly look at the structure of the discrete systems arising in the implicit transport solver. The last section of the chapter discusses the numerical errors that will result from specific choices of discretization and solution strategy.

For single-phase, incompressible flow, we have already seen that there are essentially three effects that determine the direction of the flow field. The first is heterogeneity (i.e., spatial variations in permeability) that affects the local magnitude and direction of the flow field. The second effect is introduced by drive mechanisms such as wells and boundary conditions that determine where fluids flow to and from. However, the further you are from the location of a well or a boundary condition, the less effect it will have on the flow direction. The last effect is gravity. Multiphase flow is more complicated since the fluid dynamics now also is affected by the viscosity and density ratios of the fluids present, as well as the relative permeability and capillary pressure. These effects will introduce several challenges. If the displacing fluid is more mobile than the resident fluid, it will tend to move rapidly into this fluid, giving weak shock fronts and long rarefaction waves. For a homogeneous reservoir, this will result in early breakthrough of the displacing fluid and slow and incremental recovery of the resident fluid. In a heterogeneous reservoir, one may also observe viscous fingering, which essentially means that the displacing fluid will move unevenly into the resident fluid. This is a self-reinforcing effect that will cause the fingers to move farther into the resident fluid. Gravity segregation, on the other hand, will cause fluids having different density to segregate, and lead to phenomena such as gravity override in which a lighter fluid moves quickly on top of a denser fluid. This is a problem in many recovery methods that rely on gas injection and for geological storage of CO₂. Finally, we have capillary effects that will tend to spread out the interface between the displacing and the resident fluid. When combined with heterogeneity, these effects are generally difficult to predict without detailed simulations. Sometimes, they work in the same direction to aggravate sweep and displacement efficiency, but the effects can also counteract each other to cancel undesired behavior. Gravity and capillary forces, for instance, may both reduce viscous fingering that would otherwise give undesired early breakthrough.

In this section, we will discuss most of the phenomena outlined above in more detail. In most cases, we will only consider a single effect at the time. Throughout the examples, you will also learn how to set up various types

of simulations using MRST. However, complete codes will as a rule not be discussed for brevity, and when reading the material, you should therefore take the time to also read the accompanying codes that can be found in the `in2ph` directory of the `book` module. As in most of the chapters so far in the book, you will gain much more insight if you run these codes and try to modify them to study the effect of different parameters and algorithmic choices. I also encourage you strongly to as many as possible of the computer exercises that are given throughout the text.

10.3.1 Buckley–Leverett displacement

As a first example, let us revisit the 1D horizontal setup from Example 9.3 on page 292, in which we compared the explicit and implicit solvers for the classic Buckley–Leverett displacement profile arising when pure water is injected into pure oil. The following code sets up a slightly rescaled version of the problem, computes the pressure solution, and then uses the explicit transport solver to evolve the saturations forward in time

```
G      = computeGeometry(cartGrid([100,1]));
rock  = makeRock(G, 100*milli*darcy, 0.2);
fluid = initSimpleFluid('mu' , [1, 1].*centi*poise, ...
                      'rho' , [1000, 1000].*kilogram/meter^3, 'n' , [2,2]);
bc    = fluxside([], G, 'Left', 1, 'sat', [1 0]);
bc    = fluxside(bc, G, 'Right', -1, 'sat', [0 1]);
hT   = computeTrans(G, rock);
rSol = initState(G, [], 0, [0 1]);
rSol = incompTPFA(rSol, G, hT, fluid, 'bc', bc);
rSole = explicitTransport(rSol, G, 10, rock, fluid, 'bc', bc, 'verbose',true);
```

The explicit solver uses 199 time steps to reach time 10. Let us try to see if we can do this in one step with the implicit solver:

```
[rSoli, report] = ...
implicitTransport(rSol, G, 10, rock, fluid, 'bc', bc, 'Verbose', true);
```

This corresponds to running the solver with a CFL number of approximately 200, and from the output in Figure 10.2 we see that this is not a big success. With an attempted time step of $\Delta t = 10$, the solver only manages to reduce the residual by a factor 2.5 within the allowed 25 iterations. Likewise, when the time step is halved to $\Delta t = 5$, the solver still only manages to reduce the residual one order of magnitude within the 25 iterations. Then, when the time step is halved once more, the solver converges in 20 iterations in the first step and then in 9 iterations in the next three substeps. Altogether, more than half of the iteration steps (50 out of 97) were wasted.

To overcome the problem with wasted iterations, we can subdivide the pressure step into multiple saturation steps as is done internally in the explicit solver and likewise in the implicit solver when time steps are chopped:

Time interval (s)	iter	relax	residual	rate
[0.0e+00, 1.0e+01]:	1	1.00	7.82670e+00	NaN
[0.0e+00, 1.0e+01]:	2	0.12	6.81721e+00	0.07
:	:	:	:	:
[0.0e+00, 1.0e+01]:	25	0.03	3.17993e+00	1.00
<hr/> Reducing step <hr/>				
[0.0e+00, 5.0e+00]:	1	1.00	7.03045e+00	NaN
:	:	:	:	:
[0.0e+00, 5.0e+00]:	25	0.25	6.34074e-01	0.74
<hr/> Reducing step <hr/>				
[0.0e+00, 2.5e+00]:	1	1.00	5.64024e+00	NaN
:	:	:	:	:
[0.0e+00, 2.5e+00]:	20	1.00	1.23535e-08	1.91
<hr/> Next substep <hr/>				
[2.5e+00, 5.0e+00]:	1	0.25	3.59382e-01	NaN
:	:	:	:	:
[2.5e+00, 5.0e+00]:	9	1.00	1.71654e-08	1.86
<hr/> Next substep <hr/>				
[5.0e+00, 7.5e+00]:	1	0.25	2.67914e-01	NaN
:	:	:	:	:
[5.0e+00, 7.5e+00]:	9	1.00	8.49299e-09	1.94
<hr/> Next substep <hr/>				
[7.5e+00, 1.0e+01]:	1	0.25	2.57462e-01	NaN
:	:	:	:	:
[7.5e+00, 1.0e+01]:	9	1.00	8.74041e-11	2.01
Iterations : 47			Wasted iterations : 50	
Sub steps : 4			Failed steps : 2	
Final residual : 8.74e-11			Convergence rate : 1.9	

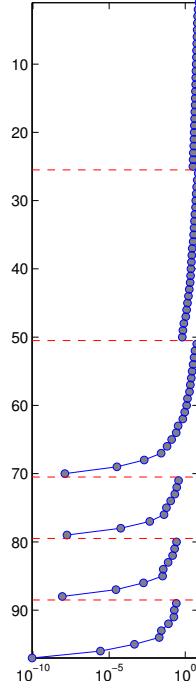


Fig. 10.2. Results from running the `implicitTransport` solver on a 1D Buckley–Leverett displacement problem with CFL number 200. Left: screen output, where several lines have been deleted for brevity. Right: convergence history for the residual, with cumulative iteration number increasing from top to bottom.

```
rSolt = rSol;
for i=1:n
    rSolt = implicitTransport(rSolt, G, 10/n, rock, fluid, 'bc', bc);
end
```

Figure 10.3 reports the approximate solutions and the overall number of iterations used by the implicit solver with n equally spaced substeps. The number of iterations is not an exact multiple of the number of time steps, since the solver typically needs more iterations during the initial time steps when the displacement front is relatively sharp. As the simulation progresses, the shock is smeared across multiple cells, which contributes to reduce the nonlinearity of the discrete system. We also see that to get comparable accuracy as the explicit transport solver, the implicit solver needs to use at least 40 time steps, which amounts to more than 160 iterations. In this case, there is thus a clear advantage of using the explicit transport solver if we want to maximize accuracy versus computational cost.

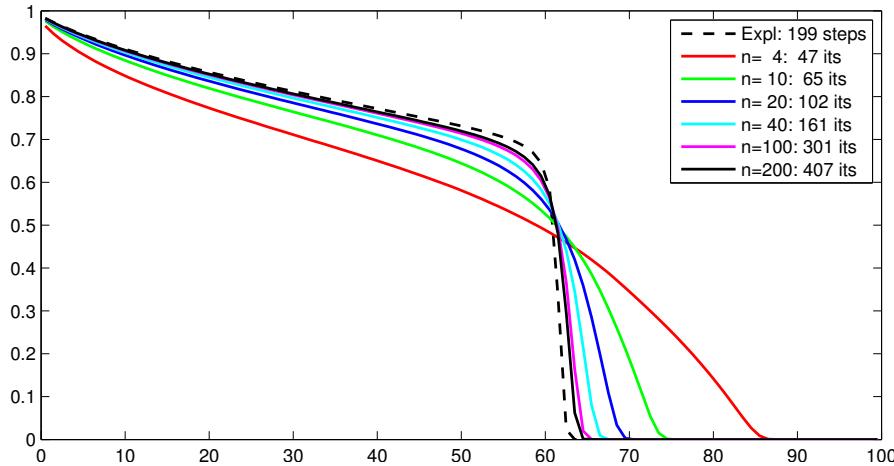


Fig. 10.3. Approximate solutions computed by the explicit transport solver and the implicit transport solver with n time steps.

10.3.2 Inverted gravity column

In the next example, we will revisit the inverted gravity column from Example 8.3 on page 273 with a light fluid at the bottom and a heavier fluid at the top. We will change the example slightly so that the fluids are representative for supercritical CO₂ and brine found at conditions that would be plausible when storing CO₂ in a deep saline aquifer. The following is the essence of the simulator (plotting commands are not included for brevity):

```

gravity reset on
G      = computeGeometry(cartGrid([1, 1, 40], [1, 1, 10]));
rock   = makeRock(G, 0.1*darcy, 1);
fluid  = initCoreyFluid('mu' , [0.30860, 0.05664]*centi*poise, ...
                      'rho' , [975.86,686.54]*kilogram/meter^3, ...
                      'n'  , [2,2], 'sr' , [.1,.2], 'kwm',[.2142,.85]);
hT    = computeTrans(G, rock);
xr   = initResSol(G, 100.0*barsa, 1.0); xr.s(end/2+1:end) = 0.0;
xr   = incompTPFA(xr, G, hT, fluid);
dt   = 5*day; t=0;
for i=1:150
    xr = explicitTransport(xr, G, dt, rock, fluid, 'onlygrav', true);
    t = t+dt;
    xr = incompTPFA(xr, G, hT, fluid);
end

```

In Example 8.3, the fluids had the same viscosity and hence moved equally fast upward and downward. Here, the supercritical CO₂ is much more mobile than the brine and will move faster to the top of the column than the brine

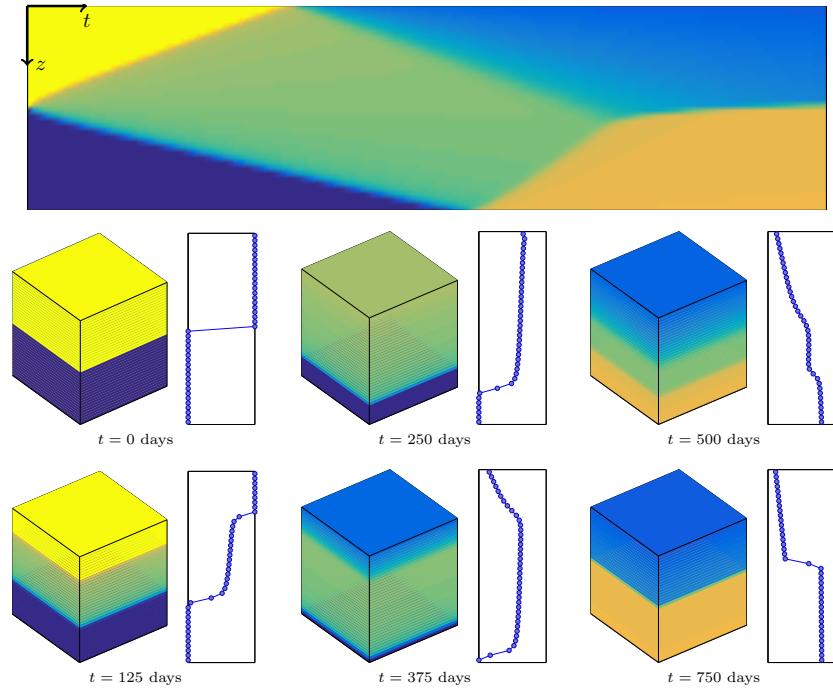


Fig. 10.4. Simulation of an inverted gravity column where pure CO₂ initially fills the bottom half and brine the upper half of the volume.

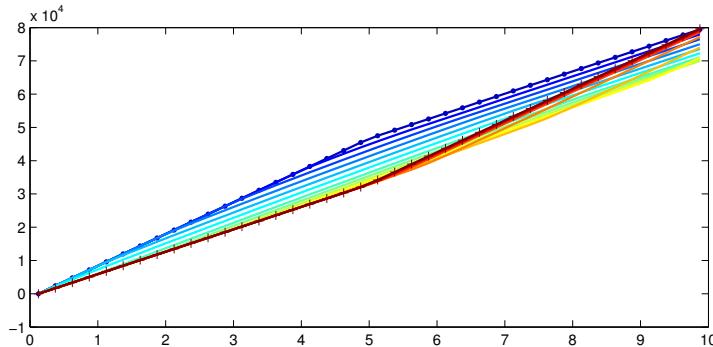


Fig. 10.5. Pressure distribution at every 10th time step as function of depth from the top of the gravity column, with blue color and marker ‘.’ indicating initial time and red color and marker ‘+’ indicating end of simulation.

moves downward. Hence, whereas the CO₂ reaches the top of the column after 250 days, it takes more than 400 days before the first brine has sunk to the bottom. After approximately two years, the fluids are clearly segregated and separated by a sharp interface.

In the simulation, we used relatively small splitting steps (150 steps of 5 days each) to march our transient solution towards steady state. Looking at Figure 10.5, which shows the vertical pressure distribution every fifty day, i.e., for every tenth time step, we see that the pressure behaves relatively smoothly compared with the saturation distribution. It is therefore reasonable to expect that we could get away with using a smaller number of splitting steps in this particular case. How many splitting steps do you think we need?

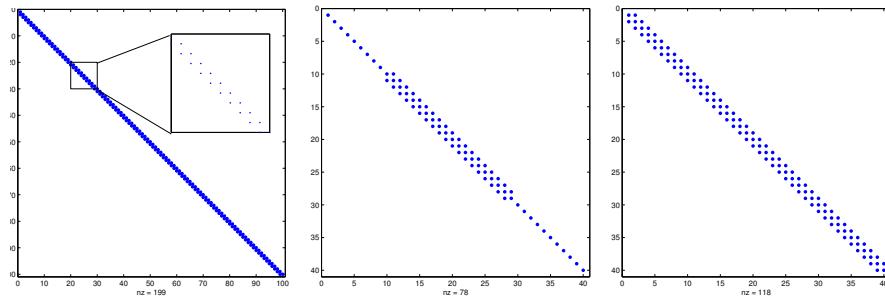


Fig. 10.6. Sparsity pattern for the 1D Buckley–Leverett problem (left) and the inverted gravity column after 125 days (middle) and 390 days (right)

Before leaving the problem, let us inspect the discrete nonlinear system arising when using the implicit transport solver in some detail. Figure 10.6 contrasts the sparsity pattern at two instances in time to that of the 1D horizontal Buckley–Leverett problem. From the discussion in Section 9.3.4, we know that the latter only has a single nonzero band below the diagonal and hence can be solved more robustly if we instead of using Newton’s method use a nonlinear substitution method with a bracketing method for each single-cell problem [186]. Since the lighter fluid moves upward and the heavier fluid moves downward during gravity segregation, there will be nonzero elements above and below the diagonal in the two-phase region and at the interface between the two phases. In the figure, we also see how the sparsity pattern changes as the two-phase region expands upward and downward from the initial interface. Since the nonlinear system is no longer triangular, a substitution method cannot be used, but each linearized system can be solved efficiently using the Thomas algorithm, which is a special $\mathcal{O}(n)$ Gaussian elimination method for tridiagonal systems. Knowing the sparsity pattern of the problem is a key to efficient solvers. The \ or mldivide solver in MATLAB performs an analysis of the linear system and picks efficient solvers for triangular, tridiagonal, and other special systems (for more info, type doc mldivide).

10.3.3 Homogeneous quarter five-spot

To gain more insight into the simulation of multiphase displacement processes we consider the classical confined quarter five-spot test case discretized on a 128×128 grid. As you may recall from Section 5.4.1, this test case consists of one quarter of a symmetric pattern of four injectors surrounding a producer (or vice versa), repeated to infinity in each direction. We assume a simplified Corey model with exponent 2.0 and zero residual saturation, i.e., $k_{rw} = S^2$ and $k_{ro} = (1 - S)^2$. We start by setting the viscosity to 1 cP for both fluid phases, giving a unit mobility ratio. We also neglect capillary and gravity forces. The injector and producer operate at fixed bottom-hole pressure, giving a total pressure drop of 100 bar across the reservoir. Since we have assumed incompressible flow, equal amounts of fluid must be produced from the reservoir so that injection and production rates sum to zero. The total time is set such that 1.2 pore volumes of fluid would be injected if the initial injection is maintained throughout the whole simulation. However, the actual injection rate will depend on the total resistance to flow offered by the reservoir, and hence vary with time when the total mobility varies throughout the reservoir as a result of fluid movement. (Remember that the total mobility will be less in all cells containing two fluid phases.) The simulation code follows the same principles as outlined in the two examples above, details can be found in the file `quarterFiveSpot2D.m`.

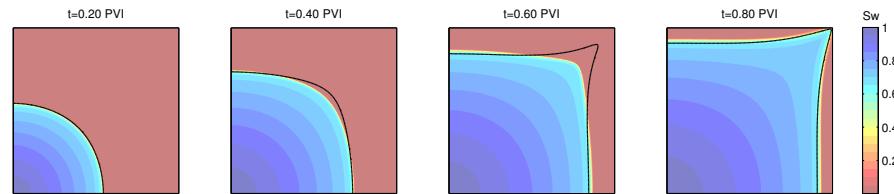


Fig. 10.7. Evolution of a two-phase displacement front for a homogeneous quarter five-spot case with water injected into oil. The single line shown in each plot is a contour at value $t/(2\sqrt{2} - 2)$ PVI for the time-of-flight field computed from the corresponding single-phase problem (i.e., with $\lambda \equiv 1$).

Figure 10.7 shows how the displacement profile resulting from the injection of water into a reservoir initially filled with oil will expand circularly near the injector. As the displacement front propagates into the reservoir, it will gradually elongate along the diagonal and form a finger that extends towards the producer. As a result, water will break through in the producer long time before the displacement front has managed to sweep the stagnant regions near the northwest and southeast corners. The evolution of the saturation profile is the result of two different multiphase effects. To better understand these effects, it is instructive to transform our 3D transport equation into streamline

coordinates. Since the flow field is incompressible, we can use (4.43) to write $\vec{v} \cdot \nabla = \phi \frac{\partial}{\partial \tau}$ so that (10.2) transforms to a family of 1D transport equations, one along each streamline,

$$\frac{\partial S}{\partial t} + \frac{\partial f_w(S)}{\partial \tau} = \frac{q_w}{\phi}. \quad (10.7)$$

Hence, the first flow effect is exactly the same Buckley–Leverett displacement as we saw in Section 10.3.1, except that it not acts along streamlines rather than along the axial directions. It is therefore tempting to suggest that to get a good idea of how a multiphase displacement will evolve, we can solve a single-phase pressure equation for the initial oil-filled reservoir, compute the resulting time-of-flight field, and then map the 1D Buckley–Leverett profile (8.59) computed from (8.52) onto time-of-flight. How accurate this approximation will be depends on the coupling between the saturation and the pressure equation. With linear relative permeabilities and unit mobility ratio, there is no coupling between pressure and transport, and mapping 1D solutions onto time-of-flight will thus produce the correct solution. In other cases, changes in total mobility will change the total Darcy velocity and hence the time-of-flight. To illustrate this, let us compare the propagation of the leading shock predicted by the full multiphase simulation and our simplified streamline analysis. For fluids with a viscosity ratio $\mu_w/\mu_n = M$, it follows by solving $f'(S) = f(S)/S$ that the leading shock of the Buckley–Leverett displacement profile would move at a speed $M/(2\sqrt{M+1}-2)$ relative to the Darcy velocity, shown as a single black line for each snapshot in Figure 10.7. Compared with our simplified streamline analysis, the movement of the injected water is retarded by the reduced mobility in the two-phase region behind the leading displacement front.

In most simulations, the primary interest is to predict well responses. To extract these, we introduce the following function

```
function wellSol = getWellSol(W, x, fluid)
    mu = fluid.properties();
    wellSol(numel(W))=struct;
    for i=1:numel(W)
        out = min(x.wellSol(i).flux,0); iout = out<0; % find producers
        in = max(x.wellSol(i).flux,0); iin = in>0; % find injectors
        lamc = fluid.relperm(x.s(W(i).cells,:))/mu; % mob in completed cell
        fc = lamc(:,1)./sum(lamc,2); %
        lamw = fluid.relperm(W(i).compi)/mu; % mob inside wellbore
        fw = lamw(:,1)./sum(lamw,2); %
        wellSol(i).name = W(i).name;
        wellSol(i).bhp = x.wellSol(i).pressure;
        wellSol(i).wcut = iout.*fc + iin.*fw;
        wellSol(i).Sw = iout.*x.s(W(i).cells,1) + iin.*W(i).compi(1);
        wellSol(i).qWs = sum(out.*fc) + sum(in.*fw);
        wellSol(i).qOs = sum(out.*(1-fc)) + sum(in.*(1-fw));
    end
```

Inside the simulation loop, this function is called as follows

```
wellSols = cell(N,1);
oip      = zeros(N,1);
for n=1:N
    x  = incompTPFA(x, G, hT, fluid, 'wells', W);
    x  = explicitTransport(x, G, dT, rock, fluid, 'wells', W);

    wellSols{n} = getWellSol(W, x, fluid);
    oip(n)      = sum(x.s(:,2).*pv);
end
```

The loop also computes the oil in place at each time step. Storing well responses in a cell array may seem unnecessary complicated and requires a somewhat awkward construction to plot the result

```
t = cumsum(dT);
plot(t, cellfun(@(x) x(2).q0s, wellSols));
```

Here, the call to `cellfun` passes elements from the cell array `wellSols` to an anonymous function that extracts the desired field containing the surface oil rate of the second well (the producer). Each element represents an individual time step. The reason for using cell arrays is to provide compatibility with the infrastructure developed for compressible models of industry-standard complexity. Here, use of cell array provides the flexibility needed to process much more complicated well output. As a direct benefit, we can use the GUI developed for visualizing well responses:

```
mrstModule add ad-core
plotWellSols(wellSols,cumsum(dt))
```

This brings up a plotting window as shown in Figure 10.8, where we visualize the surface water rate for injector and producer.

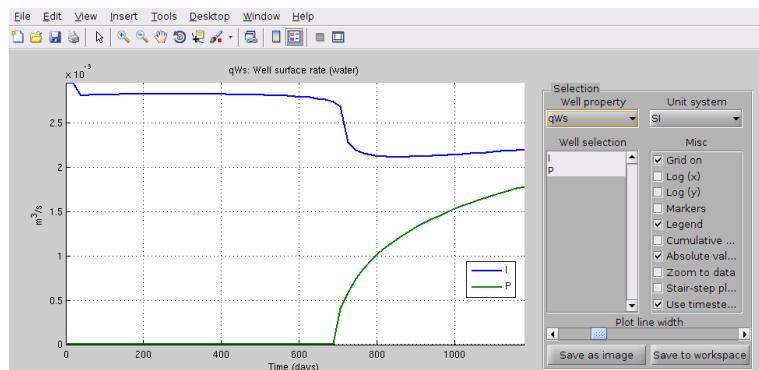


Fig. 10.8. Graphical user interface from the `ad-core` module for plotting computed well responses.

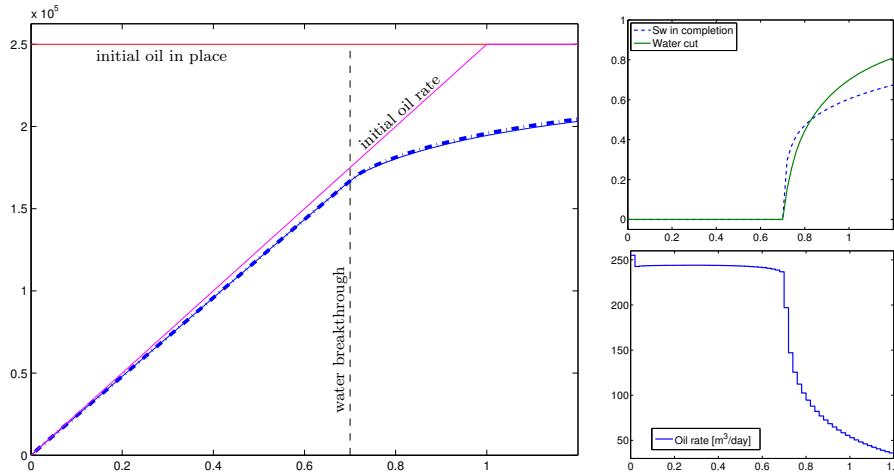


Fig. 10.9. Well responses computed for the homogeneous quarter five-spot test. The left plot shows the cumulative oil production computed from the well solution shown as a thin line compared with the amount of extracted oil derived from a mass-balance computation (initial oil in place minus current oil in place) shown as a thick dashed line. The lower-right plot shows oil rate and the upper-right plot shows water saturation and water cut (fractional flow) in the well perforation.

Let us look at the well responses shown in Figure 10.9 in more detail. We see that the oil rate drops immediately as water enters the system in the first time step and then decays slowly until water breaks through in the producer around time $t = 0.7$. Since this well now produces a mixture of water and oil, the oil rate decays rapidly as the smoothed displacement front enters the well, and then decays more slowly when the inflow of water is determined by the trailing rarefaction wave. The left plot shows the cumulative oil production computed in two different ways: (i) using the oil rate from the well solution, and (ii) measured as the difference between initial and present oil in place. Up to water breakthrough, the two estimates coincide. After water breakthrough, the production estimated from the well solution will be slightly off, since it for each time step uses a simplified approximation that multiplies the size of the time step with the total flow rate computed *at the start* of the time step and the fractional flow in the completed cell *at the end of the time step*.

COMPUTER EXERCISES:

61. Repeat the experiment with wells controlled by rate instead of pressure.
Do you observe any differences and can you explain them?
62. Repeat the experiment with different mobility ratios and Corey exponents.
63. Can you correct the computation of oil/water rates?

10.3.4 Heterogeneous quarter five-spot: viscous fingering

In the previous example we studied imbibition in a homogeneous medium, which resulted in symmetric displacement profiles. However, when a displacement front propagates through a porous medium, the combination of viscosity differences and permeability heterogeneity may introduce viscous fingering effects. In general, the term *viscous fingering* refers to the onset and evolution of instabilities at the interface between the displacing and displaced fluid phases. Fingering can arise because of viscosity differences between two phases or as a result of viscosity variations within a single phase that, for instance, contains solutes. In the laboratory, viscous fingering is usually studied in so-called Hele-Shaw cells, which consist of two flat plates separated by a tiny gap. The plates can be completely parallel, or contain small-scale variations (rugosity) to emulate a porous medium. When a viscous fluid confined in the space between the two plates is driven out by injecting a less viscous fluid (e.g., dyed water injected into glycerin), beautiful and complex fingering patterns can be observed. I highly recommend a search for “Hele-Shaw cell” on YouTube.

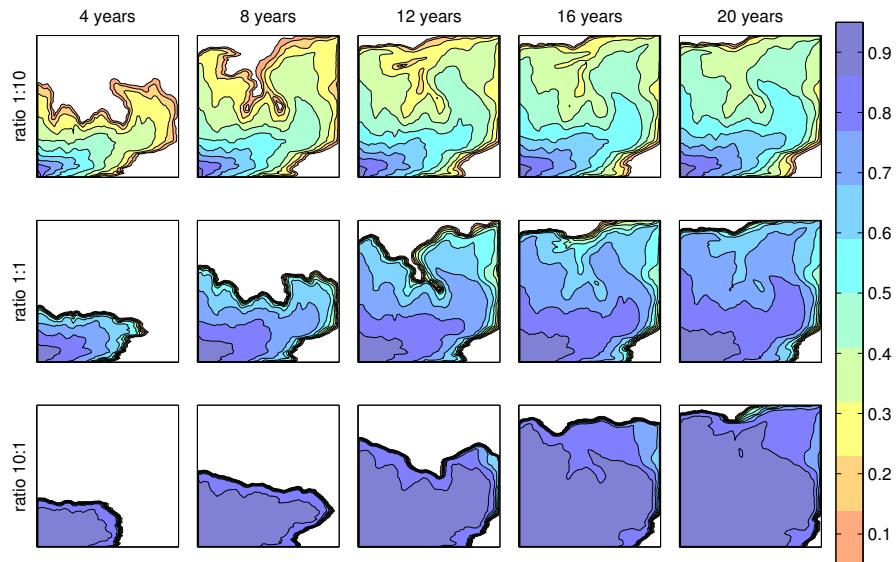


Fig. 10.10. Quarter five-spot solutions on a subsample from the first layer of the SPE 10 model for three different viscosity ratios $\mu_w : \mu_n$.

Figure 10.10 shows the simulation of a quarter five-spot on a square domain represented on a uniform 60×120 grid with petrophysical properties sampled from the topmost layer of the SPE 10 data set. The wells operate at fixed rate, corresponding to the injection of one pore volume over a period of 20 years. To reach a final time, we use 200 pressure steps and the implicit transport solver.

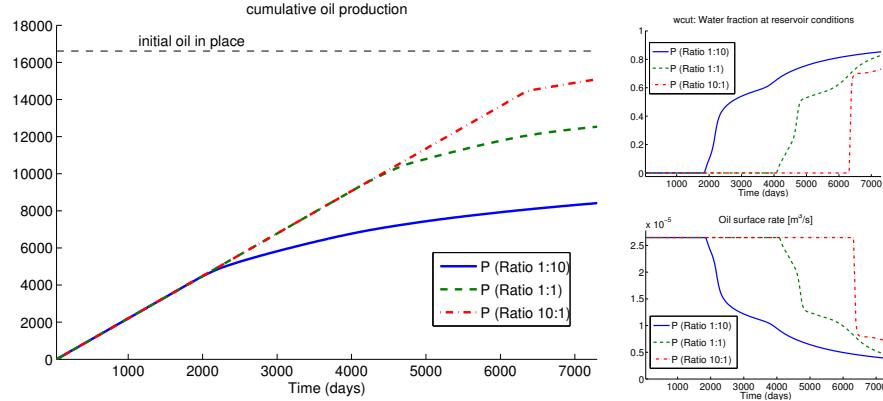


Fig. 10.11. Well responses computed for the heterogeneous quarter five-spot with different mobility ratios $\mu_w : \mu_n$.

As in the previous example, the fluid is assumed to obey a simple Corey fluid model with quadratic relative permeabilities and no residual saturations. If the viscosity of the injected fluid is (significantly) less than that of the resident fluid, we get an unfavorable displacement in which the displacing phase forms a weak shock front that will 'finger' rapidly through the less mobile phase that initially fills the medium. Once a finger develops, it will create a preferential flow direction for the injected phase, which causes the finger to extend towards the producer, following the path of highest permeability. In the opposite case of a (significantly) more viscous fluid being injected into a less viscous fluid, one obtains a strong front that acts almost like a piston and creates a very favorable and stable displacement with a leading front that has much less buckles than in the unfavorable case. Not only does this front have better local displacement efficiency (i.e., the water front can push out more oil), but the areal sweep is also better. The unit viscosity case is somewhere in between the two, having a much better local displacement efficiency than the unfavorable case, but almost the same areal sweep at the end of the simulation.

Figure 10.11 reports well responses for the three simulations. Since water is injected at a fixed rate in all simulations, the oil rate will remain constant until water breaks through in the producer. This happens after 1825 days in the unfavorable case, after 4050 days for the case with equal viscosities, and after 6300 days in the favorable case. As discussed in the previous example, the decay in oil rate depends on the strength of the displacement front and will hence be much more abrupt in the favorable mobility case, which has an almost piston-like displacement. On the other hand, by the time the favorable case breaks through, the unfavorable case has reached a water cut of 82%. Water handling is generally expensive and in the worst case the unfavorable case might have to shut down before reaching the end of the 20 year production period. (Complete code for the example is found in `viscousFingeringQ5.m`.)

Next, let us look at the sparsity structure of the discretized transport equation. In the homogeneous case discussed in the previous section, all fluxes point in the positive axial directions and hence the Jacobian matrix will be lower triangular. With a heterogeneous permeability, or another well pattern, this unidirectional flow property is no longer present and the Jacobi matrix will have elements above and below the diagonal, as seen in Figure 10.12. However, if we look at the transformation to streamline coordinates (10.7), it is clear that we still have unidirectional flow along streamlines. This means that the Jacobi matrix can be permuted to triangular form by performing a topological sort on the flux graph derived from the total Darcy velocity. In MATLAB, one can try to try to permute the matrix to upper-triangular form by use of the Dulmage–Mendelsohn decomposition

```
[p,q] = dmperm(J); Js = J(p,q);
```

The result is shown in the middle plot in Figure 10.12. This permutation is similar to what is done inside MATLAB’s linear solver `mldivide`. Since we can permute the Jacobi matrix to triangular form, we can also do the same for the nonlinear system, and hence apply a highly efficient nonlinear substitution method [186] as discussed for 1D Buckley–Leverett problems in Section 9.3.4. The same applies also for 3D cases as long as capillary forces are neglected and we have purely co-current flow. Countercurrent flow can be introduced by gravity segregation, as we saw in Section 10.3.2 above, or if the flow field is computed by one of the consistent discretization schemes from Chapter 6, which are generally not monotone.

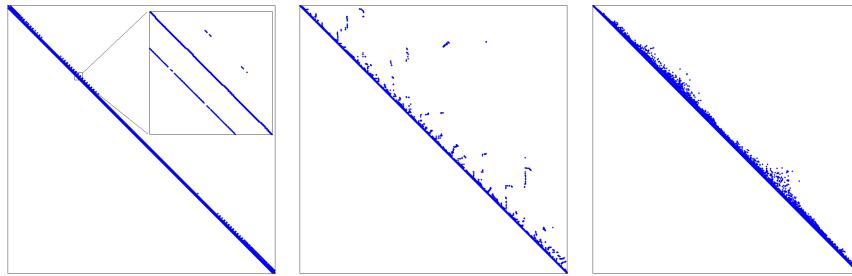


Fig. 10.12. The left plot shows the sparsity structure for the Jacobi matrix for the heterogeneous quarter five-spot with viscosity ratio 1:10. The middle plot shows the sparsity structure after a topological sort, whereas the right plot shows the sparsity structure after potential ordering.

It is also possible to permute the discretized system to triangular form by performing a potential ordering [144] as shown to the right in Figure 10.12. This is done as follows

```
[~,i] = sort(x.pressure); Jp = J(i,i);
```

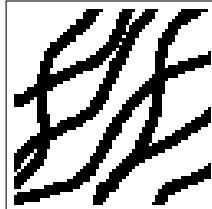
Implementing this type of nonlinear substitution methods is unfortunately not very efficient in MATLAB and should be done using a compiled language. In MRST, we therefore mainly rely on the intelligence built into `mldivide` to give us the required computational performance.

A remark at the end: in a real case, injectivity would obviously be a decisive factor, i.e., how high pressure is required to ensure a desired injection rate, or vice versa, which rate one would obtain for a given injection pressure. This is not accounted for in our discussion above; for illustration purposes we tacitly assumed that desired injection rate could be maintained.

COMPUTER EXERCISES:

64. Repeat the experiments with wells controlled by pressure, fixed water viscosity and varying oil viscosity. Can you explain the differences you observe?
65. Run a systematic study that repeats the simulation above from each of the 85 layers of the SPE 10 model. Plot and compare the resulting production curves. (The experiment can be run for a single mobility ratio to save computational time).
66. Run the same type of study with 100 random permeability fields, e.g., as generated by the simplified `gaussianField` routine from Section 2.5.2. Alternatively, you can use any kind of drawing program to generate a bitmap and generate a channelized permeability as follows

```
K = ones(G.cartDims)*darcy;
I = imread('test.pbm');
I = flipud(I(:,:,1));
K(I) = milli*darcy;
```



This can easily be combined with different random fields for the foreground and background permeability.

10.3.5 Buoyant migration of CO₂ in a sloping sandbox

In Section 10.3.2, we considered the buoyant migration of supercritical CO₂ inside a vertical column. Here, we extend the problem to three spatial dimensions and simulate the upward movement of CO₂ inside a sloping sandbox with sealing boundaries. The rectangular sandbox has dimensions 100 × 10 × 200 m³, and we will consider two different petrophysical models: homogeneous properties or Gaussian porosity with isotropic permeability given from a Carman–Kozeny transformation similar to (2.6). The sandbox is rotate around the *y*-axis so that the top surface makes an angle of inclination θ with the horizontal plane. Instead of rotating the grid so that it aligns with the geometry, we will rotate the coordinate system by rotating the gravity vector an angle θ around the *y*-axis, see Figure 10.13.

The rotation is introduced as follows,

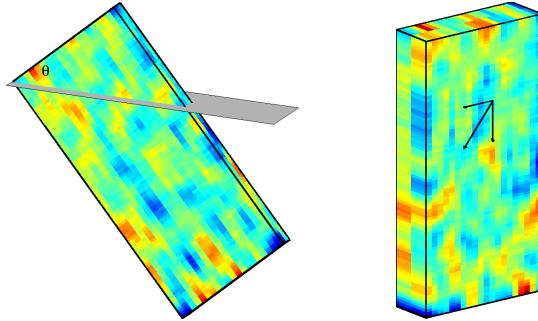


Fig. 10.13. Illustration of the sloping sandbox used for the buoyancy example and how it is simulated by rotating the gravity vector. (Color: Gaussian porosity field).

```
R = makehgtform('yrotate',-pi*theta/180);
gravity reset on
gravity( R(1:3,1:3)*gravity().'');
```

MRST defines the gravity vector as a persistent, global variable which by default equals $\vec{0}$. The second line ensures that \vec{g} is set to the standard value (pointing downward in the vertical direction) before we perform the rotation.

To initialize the problem, we assume that CO₂, which is lighter than the resident brine, fills up the model from the bottom and to a prescribed height,

```
xr = initResSol(G, 1*barsa, 1);
d = gravity() ./ norm(gravity());
dc = G.cells.centroids * d.';
xr.s(dc>max(dc)-height) = 0;
```

For accuracy and stability, the time step is ramped up gradually as follows,

```
dT = [.5, .5, 1, 1, 1, 2, 2, 2, 5, 5, 10, 10, 15, 20, ...
        repmat([25, [1,97]])).*day;
```

to reach a final simulation time of 2500 days. The remaining code is similar to what was discussed above; details can be found in `buoyancyExample.m`.

Let us consider the homogeneous case first. Initially, the buoyant CO₂ plume will form a cone shape as it migrates upward and gradually drains the resident brine. After approximately 175 days, the migrating plume starts to accumulate as a thin layer of pure CO₂ under the sloping east face of the box. This layer will migrate quickly up towards the topmost northeast corner of the box, which is reached after approximately 400 days. This corner forms a structural trap that will gradually be filled as more CO₂ migrates upward. The trapped CO₂ forms a diffused and curved interface (see the plots at 500 and 1000 days), but as time passes, the interface becomes sharper and flatter. During the same period, brine will imbibe into the trailing edge of the CO₂ plume and gradually formed a layer of pure brine at the bottom.

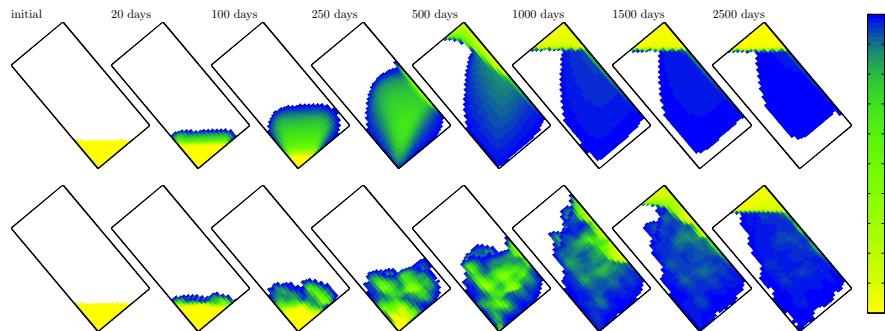


Fig. 10.14. Buoyant migration of CO₂ in a sandbox with sealing boundaries. The upper plots show the case with homogeneous case and the lower plots the case with Gaussian heterogeneity. In the plots, only cells containing some CO₂ are colored.

After approximately 1000 days, the only CO₂ left below the interface in the northeast corner is found at small saturation values and will therefore migrate very slowly upward.

The heterogeneous case follows much of the same pattern, except that the leading drainage front will finger into high-permeable regions of the sandbox. Low-permeable cells, on the other hand, will retard the plume migration. Altogether, we see a significant delay in the gravity segregation compared with the homogeneous case. Since the permeability is isotropic, the plume will still mainly migrate upward. This should not be expected in general. Many reservoirs will have significantly lower permeability in the vertical direction or consist of strongly layered sandstones containing mud drapes or other thin deposits that inhibit vertical movement between layers. For such cases, one can expect a much larger degree of lateral movement.

COMPUTER EXERCISES:

67. To gain more insight into the flow physics of a buoyant phase, you should experiment more with the script discussed above. A few examples:
 - Try to set $\theta = 88$ and the initial height to 10 meters for the heterogeneous case.
 - Set $\theta = 60^\circ$ and impose an anisotropic permeability with ratio $0.1 : 1 : 5$ to mimic a case with strong layering.
 - In the experiments above, we used an unrealistic fluid model without residual saturations. Replace the fluid model by a more general Corey model having residual saturations (typical values could be 0.1 or 0.2) and possibly also end-point scaling. How does this affect the upward plume migration? (Hint: in addition to the structural trapping at the top of the formation, you will now have residual trapping.)
68. Rerun the experiment with capillary forces included, e.g., by using `initSimpleFluidJfunc` with extra parameters set as in its documentation.

69. Instead of using an initial layer of CO₂ at the bottom of the reservoir, try to introduce an injector that injects CO₂ at constant rate before it is shut in. For this, you should increase the spatial and temporal scales of the problem
70. In cases like this, gravity will introduce circular currents that destroy the unidirectional flow property we discussed in some of the previous examples. To investigate the sparsity of the discretized transport equations, you can set a breakpoint inside the private function `newtonRaphson2ph` used by the implicit transport solver and use the `plotReorder` script from the `book` module to permute the Jacobi matrix to block-triangular form. Try to stop the simulation in multiple time steps to investigate how the sparsity structure and degree of countercurrent flow change throughout the simulation.

10.3.6 Water coning and gravity override

Water coning is a production problem in which water (from a bottom drive) is sucked up in a conical shape towards a producer. This is highly undesirable since it reduces the hydrocarbon production. As an example, we consider a production setup on a sector model consisting of a two different rock types separated by a fully conductive, inclined fault as shown in Figure 10.15. A vertical injector is placed the low-permeable stone ($K=50$ md and $\phi = 0.1$) to the east of the fault, whereas a horizontal producer is perforated along the top of the more permeable rock ($K=500$ md and $\phi = 0.2$) to the west of the fault. The injector operates at a fixed bottom-hole pressure of 700 bar and the producer operates at a fixed bottom-hole pressure of 100 bar. To clearly illustrate the water coning, we consider oil with somewhat contrived properties: density 100 kg/m³ and viscosity 10 cP. The injected water has density 1000 kg/m³ and viscosity 1 cP. Both fluids have quadratic relative permeabilities. The large density difference was chosen to ensure a bottom water drive, while the high viscosity was chosen to enhance the coning effect. Complete source code can be found in `coningExample.m` in the `book` module.

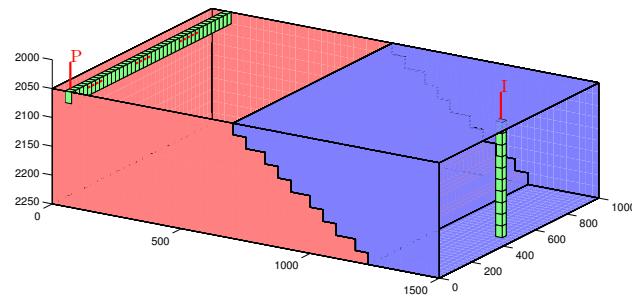


Fig. 10.15. Sector model used to demonstrate water coning. Blue color indicates low permeability and red color high permeability.

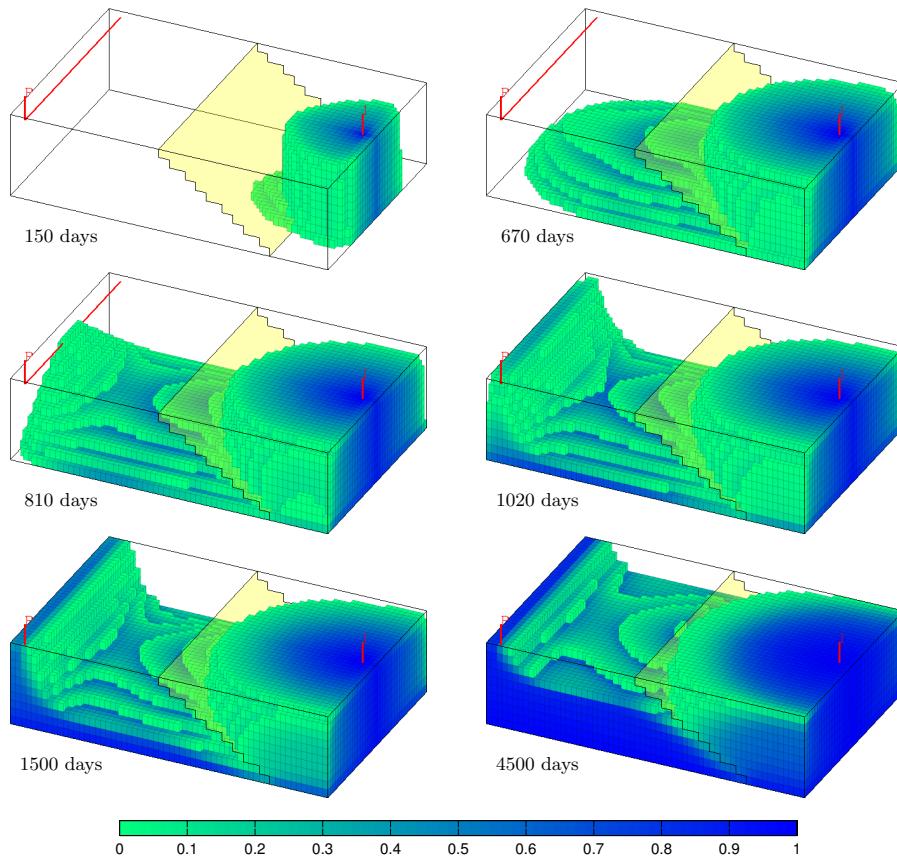


Fig. 10.16. Evolution of the displacement profile for the water-coning case.

The idea of using a vertical injector is that the lower completions will set up a bottom water drive in the good rock to the west of the fault, whereas the upper completions will provide volumetric sweep of the low-quality rock to the east of the fault. The water front from the lowest perforations penetrates through to the better zone west of the fault after approximately 40 days and then gradually builds up a water tongue that moves westward more rapidly along the bottom of the reservoir as seen in the two plots in the upper row of Figure 10.16. The advancing water front reaches the far west side of the reservoir after approximately 670 days and forms a cone that extends upward towards the horizontal producer. After 810 days, the water front breaks through in the mid-section of the producer and after 1020 days, the whole well is engulfed by water. If the well had been instrumented with intelligent inflow control devices, the operator could have reduced the flow rate through the mid-section of the well to try to delay water breakthrough. As the water is sucked up to the producer, it gradually forms a highly conductive pathway

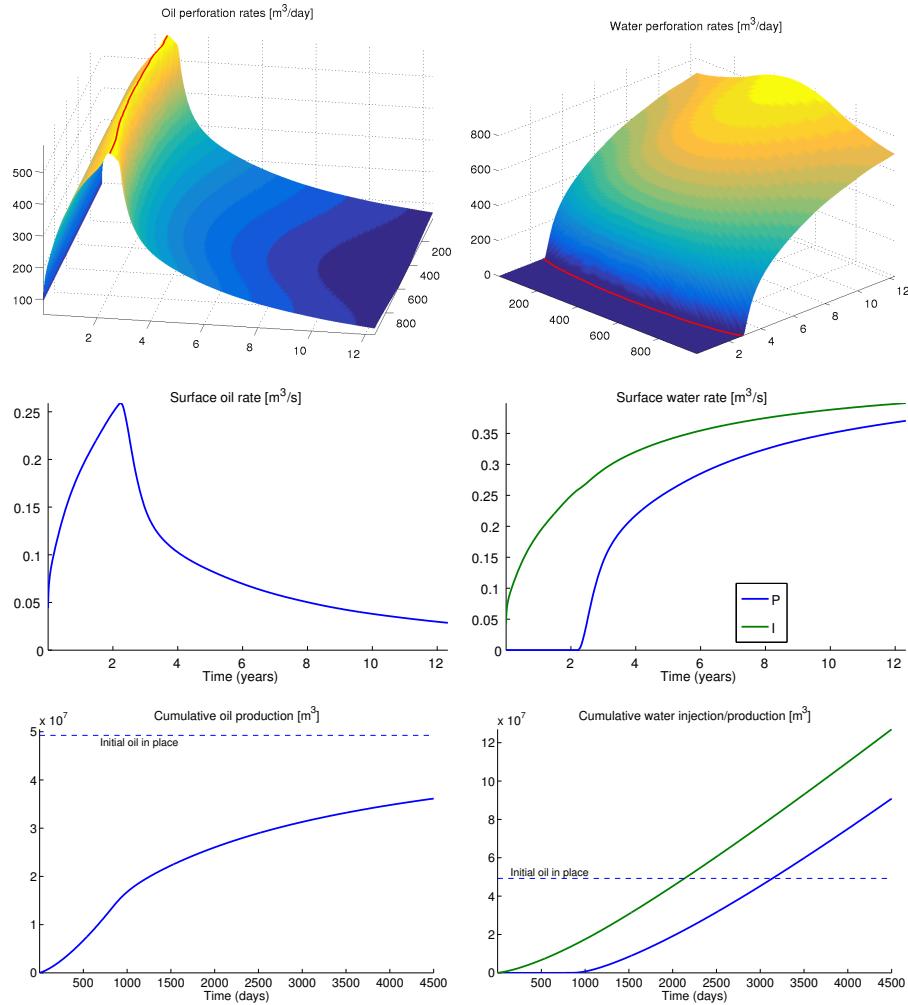


Fig. 10.17. Well responses for the simulation of water coning. The top plots show rates in each perforation of the horizontal producer; the red lines indicate water breakthrough. The middle plots show total surface rates, whereas the lower plots show cumulative oil production and cumulative water injection and production.

from the injector to the producer as seen in the snapshot from time 1500 days shown at the bottom-left of Figure 10.16. A significant fraction of the injected water will therefore cycle through the water zone without contributing significantly to sweep any unproduced oil, which will contribute to significantly increase the energy consumption and the operational costs of the production operation.

Looking at the well responses in Figure 10.17, we first of all observe that the initial injection rate is very low because of the high viscosity of the resident oil. Thus, we need a quite high injection pressure to push the first water into the reservoir. Once this is done, the injectivity increases steadily as more water contacts and displace a fraction of the oil. Since the reservoir rock and the two fluids are incompressible, increased injection rates will give an equal increase in oil production rates until water breaks through after approximately 800 days. Since there is no heterogeneity to create pockets of bypassed oil, and residual saturations are zero, we will eventually be able to displace all oil by continuing to flush the reservoir with water. However, the oil rate drops rapidly after breakthrough and increasing amounts of water need to be cycled through the reservoir to wash out the last parts of the remaining oil. By 4500 days, the recovery factor is 73% and the total injected and produced water amount to approximately 2.6 and 1.8 pore volumes, respectively.

Another related problem is that of gravity override in which a less dense and more mobile fluid flows preferentially above a denser and less mobile fluid. To illustrate this multiphase flow phenomenon, we consider a reservoir that consists of two horizontal zones placed on top of each other. Light fluid of high mobility is injected into the lower zone by a vertical well placed near the east side. Fluids are produced from a well placed near the west side and perforated in the lower zone only, see Figure 10.18. Densities of the injected and resident fluids are assumed to be 700 kg/m^3 and 1000 kg/m^3 , respectively. To accentuate the phenomenon, we assume that both fluids have quadratic relative permeabilities, zero residual saturations, and viscosities 0.1 cP and 1.0 cP, respectively. In the displacement scenario, 0.8 pore volumes of the light fluid are injected at constant rate. For simplicity, we will refer to this fluid as water and the resident fluid as oil.

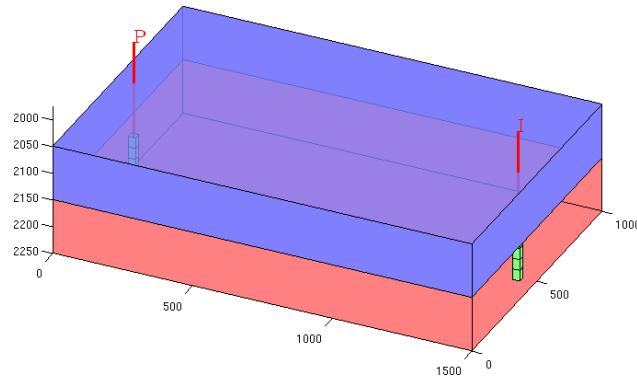


Fig. 10.18. Setup for the case used to illustrate gravity override. Blue and red color indicate different permeabilities. Perforated cells are colored green.

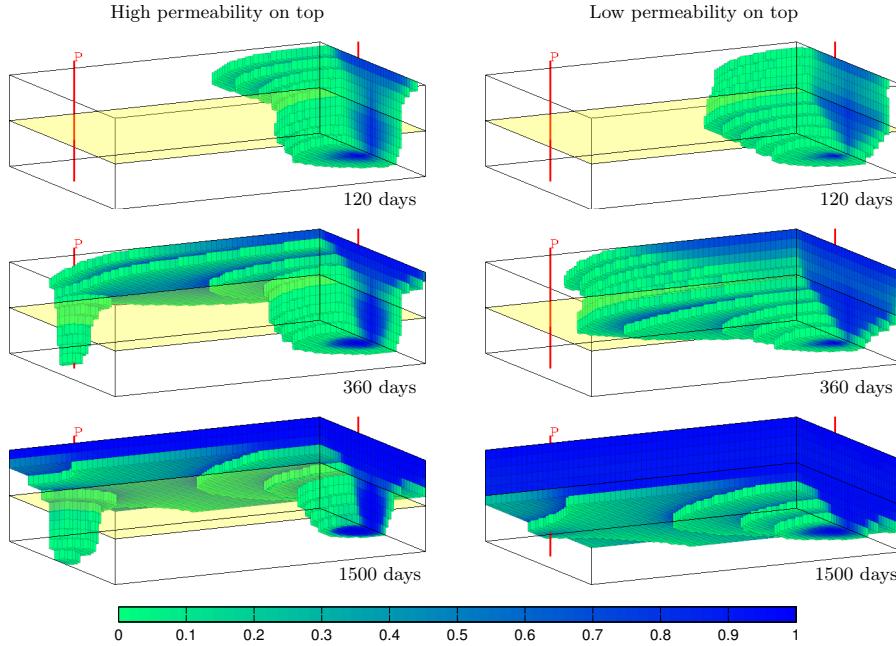


Fig. 10.19. Evolution of the displacement profiles for the gravity-override setup. The reservoir is viewed from a position below and to the southeast of the reservoir. Cells containing only the resident fluid are not plotted.

We consider two different scenarios: one with high permeability in the upper zone and low permeability in the lower zone, and one with low permeability in the upper zone and high permeability in the lower zone. To simplify the comparison, the injection rate is the same in both cases. Figure 10.19 compares the evolving displacement profiles for the two cases. In both cases, buoyancy will quickly cause the lighter injected fluid to migrate into the upper zone. With high permeability at the top, the injected fluid will accumulate under the sealing top and flow fast towards the west boundary in the upper layers, as seen in the upper-left plot of Figure 10.19. Looking at Figure 10.20, you may observe that since the production well is pressure-controlled and perforated in the low-permeable zone, the perforation rates will increase towards the top, i.e., the closer the perforation lies to the high-permeable zone above. As the leading displacement front reaches the producer near the west boundary, it is sucked down towards the open perforations and engulfs them almost instantly, as seen after 360 days in the middle-left plot of Figure 10.19 and the red line in the lower-left plot of Figure 10.20. This causes a significant drop in the oil fluid and a corresponding increase in water rates towards the top of the well, which lies closer to the flooded high-permeable zone. The oil rate is also reduced in the lowest perforations, but since these layers do not produce much of the injected fluid, the drop in oil rate is an effect of the internal adjustment

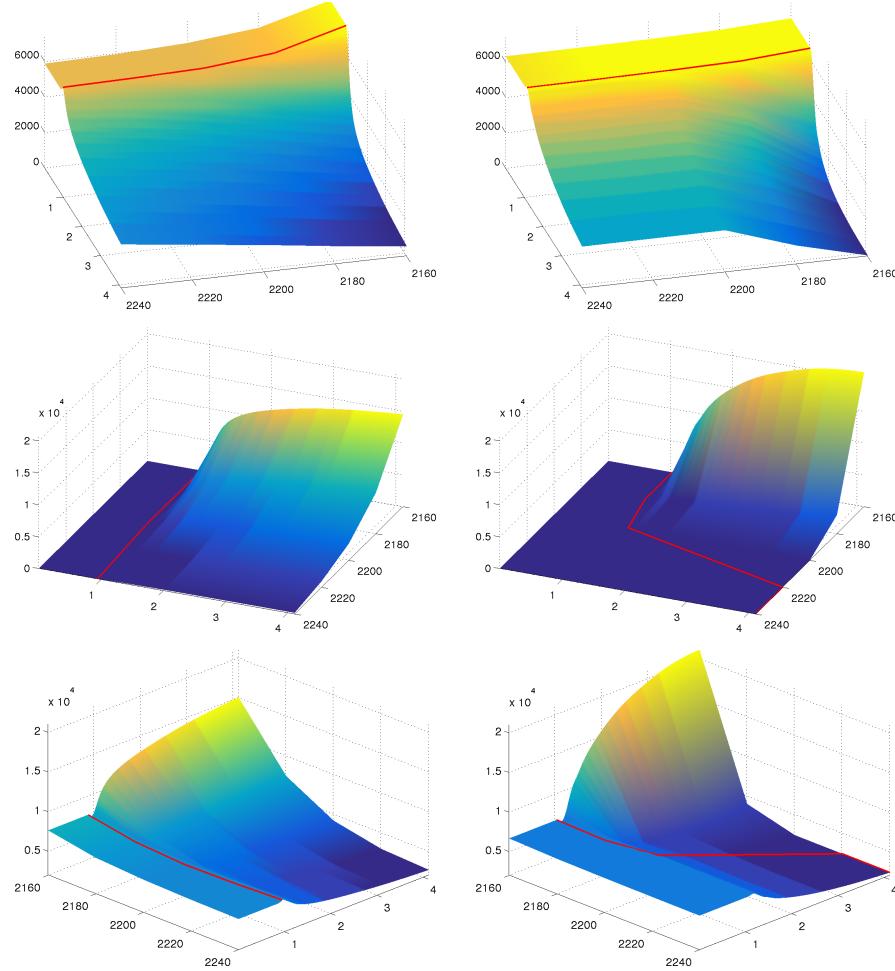


Fig. 10.20. Perforation rates for the gravity-override case: oil (top), water (middle), and total rate (bottom). For the oil rate, the red lines indicate peak production, while they indicate water breakthrough in the plots of water and total rate. The left column reports the case with a high-permeability upper zone and the right column the opposite case.

of total perforation rates along the well since mobility is so much higher near the top of the well. Indeed, some time after water breakthrough the oil rate is higher in the lower than in the upper perforations.

Also in the case with a low-permeable zone on top, buoyancy will cause the injected fluid to migrate relatively quickly into the upper zone. However, since this case has better direct connection between the injector and producer through the high-permeable lower zone, the displacement front will move relatively uniformly through all layers of the top zone and the upper layer of the

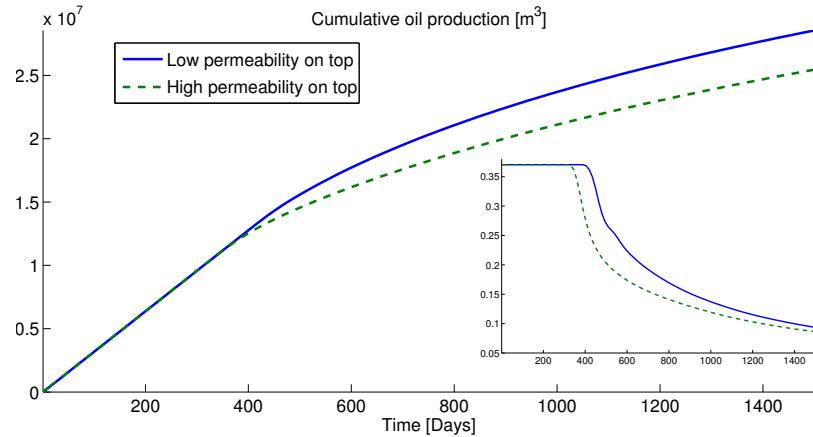


Fig. 10.21. Cumulative oil production for the gravity-override case. The inset shows surface oil rates.

lower zone. The leading part of the displacement front is therefore both higher and wider than in the other case and has swept a larger part of the upper zone by the time it breaks through in the producer. Unlike in the first case, the displacing fluid will not engulf the whole well, but only breaks through in the three topmost perforations. However, after breakthrough, the majority of the production comes from the topmost perforation, and the total rate in the three lowest perforations quickly drops below 20% of the rate in the topmost perforation.

Finally, looking at cumulative oil production and the surface oil rates reported in Figure 10.21, we see that having a low permeability on top not only delays the water breakthrough, but also enables us to maintain a higher oil rate for a longer period. This case is more economically beneficial than the case with high permeability on the top. To increase the recovery of the latter case, one possibility would be to look for a substance to inject with the displacing fluid to reduce its mobility in the upper zone.

10.3.7 The effect of capillary forces – capillary fringe

As you may recall from Section 8.1.3, the pressure in a non-wetting fluid is always greater than the pressure in the wetting phase. In a reservoir simulation model, the capillary pressure – defined as $p_c = p_n - p_w$ in a two-phase system – has the macroscale effect of determining the local saturation distribution at the interface between the wetting and non-wetting fluid, or in other words, there is a relation $p_c = P_c(S)$ between capillary pressure and saturation. This is seen in two ways: In a system that is initially in hydrostatic equilibrium, capillary forces enforce a smooth, vertical transition in saturation upward from a (horizontal) fluid contact. This transition is often referred to as the

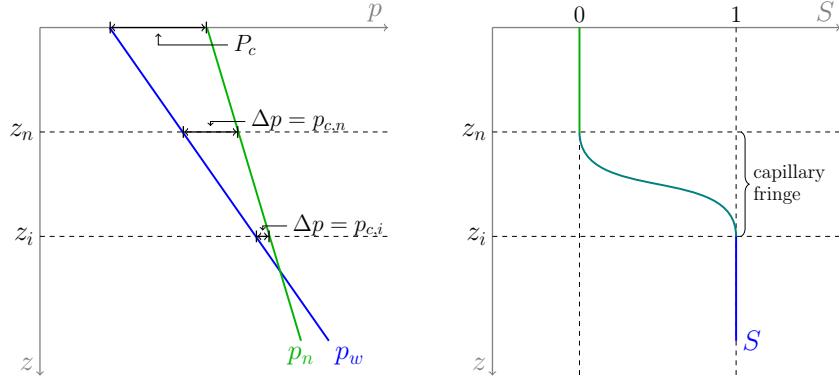


Fig. 10.22. Diagrams showing phase pressures and capillary pressure (left) and saturation (right) as function of depth. Here, z_i is the depth of the contact between the non-wetting and the wetting fluid and z_n is the depth of pure non-wetting fluid.

capillary fringe, and will be discussed in more detail in this section. The second effect is that capillary forces will redistribute fluids slightly near a dynamic displacement front so that this is not a pure discontinuity, as assumed in the hyperbolic models discussed in Chapter 9, but rather a smooth wave. For many field and sector models, the characteristic width of the transition zone is small compared to the typical grid size and hence capillary forces can be safely neglected. In other cases, capillary forces have a significant dampening effect on the tendency for viscous fingering and is therefore crucial to include in the simulation model.

Returning to the formation of a capillary fringe near a fluid interface, let z_i denote the depth of the contact between the wetting and non-wetting fluid, and let $p_{n,i}$ and $p_{w,i}$ denote the phase pressures at this depth. The phase pressures and the capillary pressure is then given by,

$$\begin{aligned} p_w(z) &= p_{w,i} + g\rho_w(z - z_i), & p_n(z) &= p_{n,i} + g\rho_n(z - z_i) \\ p_c(z) &= p_{c,i} + g\Delta\rho(z - z_i) \end{aligned} \quad (10.8)$$

where $\Delta\rho = \rho_n - \rho_w$ is the density difference and $p_{n,i}$ is the capillary pressure at z_i . This pressure is the capillary pressure that is necessary to initiate displacement of the wetting fluid by the non-wetting fluid and is called the *entry pressure*. It follows from (10.8), that the total height of the capillary fringe is given by $p_{c,n}/g\Delta\rho$. Figure 10.23 illustrates the concept of a capillary fringe for a case with zero residual saturations.

If we know the phase contact z_i , the saturation can be found directly by first computing the capillary force as function of depth using (10.8) and then using the capillary pressure function $P_c(S)$ to invert for saturation. Alternatively, if we only know the volume of the fluids, we can use the incompressible solvers to determine the hydrostatic fluid distribution. To illustrate, we consider a $100 \times 100 \text{ m}^2$ vertical cross-section represented on a 20×40 grid. We

will assume a fluid system with CO₂ and brine having the same basic properties as in Section 10.3.2. With a density difference of approximately 290 kg/m³, the capillary fringe corresponding to a capillary pressure of 1 bar has height 35 m. To model that the relationship between saturation and capillary pressure depends on permeability and porosity, we use the Leverett J-function (8.9). The `initSimpleFluidJfunc` implements a simplified Corey-type fluid in which $J(S) = 1 - S$. This gives,

$$P_c(S) = \sigma \sqrt{\frac{\phi}{K}}(1 - S).$$

The surface tension σ is usually specific to the fluid, but to illustrate the effect of capillary raise, we choose a value such that median rock properties give a capillary pressure of one bar,

```
fluid = initSimpleFluidJfunc('mu' , [0.30860, 0.056641]*centi*poise, ...
    'rho', [ 975.86,  686.54]*kilogram/meter^3, ...
    'n' , [      2,         2], ...
    'surf_tension',1*barsa/sqrt(mean(rock.poro)/(mean(rock.perm))),...
    'rock',rock);
```

We set initial data such that the rock is filled with half a pore volume of wetting fluid at the bottom and half a pore volume of non-wetting fluid at the top. We then simulate the system forward in time until steady-state is reached. The time-loop is set up with a gradual ramp-up of the time step as follows to increase the stability of the solution procedure (we will come back to the choice of time step in Section 10.4.1),

```
dt = dT*[1 1 2 2 3 3 4 4 repmat(5,[1,m])]*year;
dt = [dt(1).*2.^[-5:-5:1:-1], dt(2:end)];
s = xr.s(:,1);
for k = 1 : numel(dt),
    xr = incompTPFA(xr, G, hT, fluid);
    xr = implicitTransport(xr, G, dt(k), rock, fluid);
    t = t+dt(k);
    if norm(xr.s(:,1)-s,inf)<1e-4, break, end;
end
```

We consider two different permeability setups. In the first case, the permeability field varies linearly from 50 md in the west to 400 md in the east. The porosity is assumed to be constant. When the system is released from the artificial initial state with a sharp interface at $z = 50$, the non-wetting fluid starts draining downward into the wetting phase whereas the wetting phase starts imbibing upward into the non-wetting phase. After approximately 3.5 years, the system reaches the steady state shown in the middle plot in the upper row of Figure 10.23. Here, we say that steady-state is reached when the saturation difference between two consecutive time steps is less than 10^{-4} measured in the L[∞] norm. Because the permeability is homogeneous in the

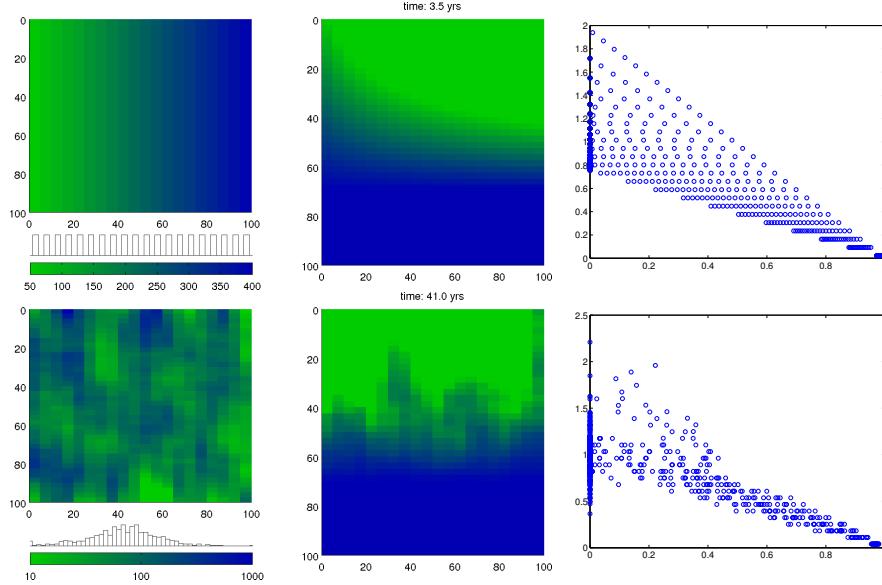


Fig. 10.23. Capillary fringe for two different permeability fields: permeability increasing linearly from west to east (top), and lognormal permeability (bottom).

vertical direction, the steady-state saturation decreases linearly upward from pure wetting to pure non-wetting fluid. The height of the capillary fringe is much higher in west where the permeability is low, since the capillary scales as $1/\sqrt{K}$, and lower in the east where the permeability is high. In the plot of capillary pressure versus saturation to the upper-right in Figure 10.23, you may also be able to identify the twenty different lines corresponding to the twenty columns of homogeneous permeability in the grid.

The second case has random petrophysical parameters with a permeability field that is related to a Gaussian porosity field through a Carman–Kozeny relationship, see the lower-left plot in Figure 10.23. The permeability values span three orders of magnitude, from 1 md to 1 darcy, which in turn gives a wider span in time constants than in the previous case. Likewise, as in the previous case, the imbibing wetting phase will be sucked higher up and sideways into regions of lower permeability. If you run the script `capillaryColumn` yourself, you will see that the high-permeable regions surrounding the initial sharp interface reach equilibrium within a few years, whereas the rise is much slower in the low-permeable regions, in particular in the column next to the east boundary, which is the last to reach steady state, after approximately 41 years. At steady state, the fringe extends above the top of the reservoir section in the east-most column. Moreover, since the permeability is heterogeneous, the saturation at steady state is no longer monotone in the vertical (and horizontal) direction.

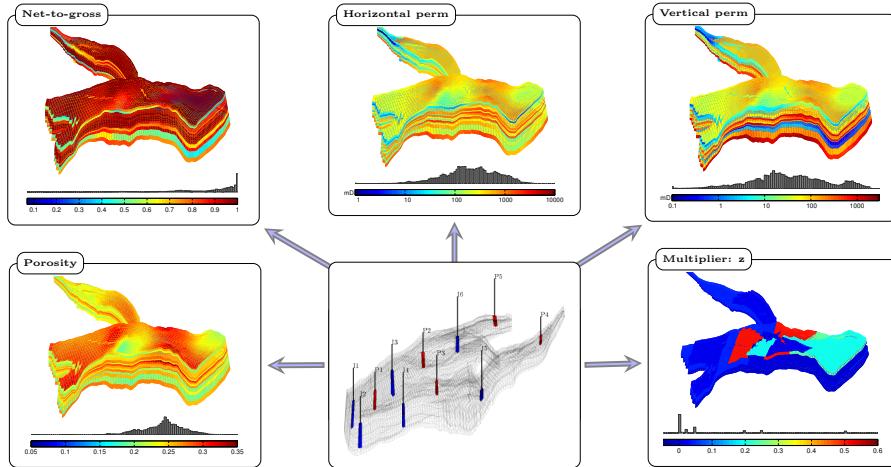


Fig. 10.24. The Norne test case with grid and petrophysical data from the simulation model of the real field. The well pattern is artificial and has nothing to do with the real model.

10.3.8 Norne: simplified simulation of a real-field model

Having worked with highly idealized models so far in this chapter, it is now time to look at a more realistic model. For this, we will use the grid geometry and the petrophysical properties from a simulation of the Norne field from the Norwegian Sea. More details about Norne and the reservoir geometry were given in Section 3.3.1. Figure 10.24 shows the petrophysical properties as well as a well-pattern that was chosen somewhat haphazardly for illustration purposes. We notice that the permeability is anisotropic and heterogeneous, with a clear layered structure. This layered structure is also reflected in the histograms, which show several modes. (Such histograms are discussed in more detail in Sections 2.5.3 and 2.5.5 for the SPE 10 and SAIGUP models). The lateral permeability has four orders of magnitude variations, while the vertical permeability is up to two orders lower and has five orders of magnitude variations. The histogram plot the histograms as discussed for the SPE 10 and SAIGUP models in .) In addition, the vertical communication is further reduced by a multiplier field (`MULTZ` keyword), which contains large regions having values close to zero in the middle layers of the reservoir. The porosities span the interval [0.094,0.347], but since the model has a net-to-gross field to model that a portion of the cells may consist of impermeable shale, the effective porosity will be much smaller in some of the cells. For a model like this, we thus cannot expect to be able to use the explicit transport solver and must instead rely on the implicit solver.

Setting up the model proceeds in a similar fashion as discussed previously, and the interested reader can consult the `runNorneSimple` script in the `book` module (a similar example with synthetic petrophysical properties can be

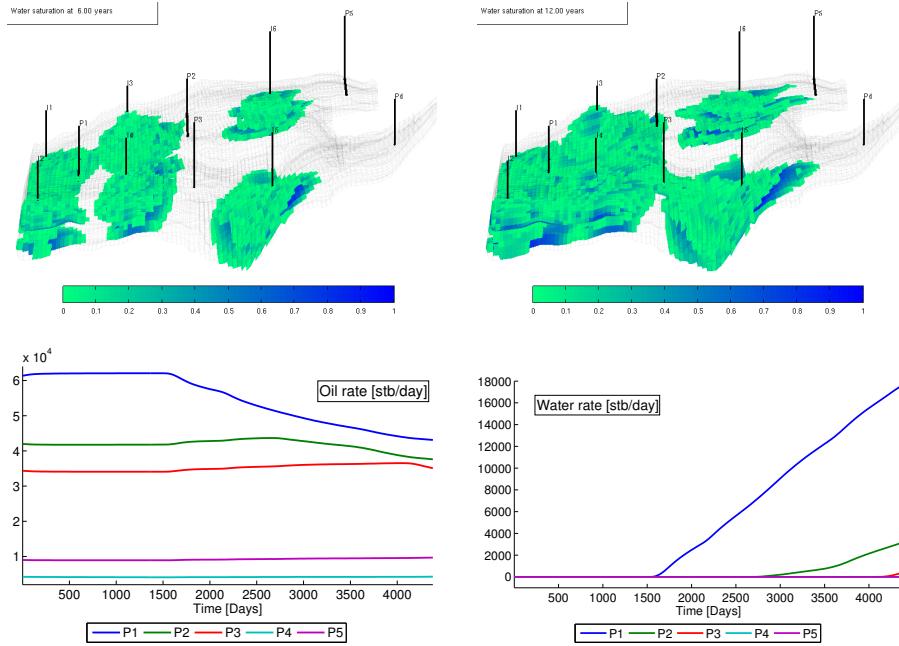


Fig. 10.25. Incompressible two-phase simulation for the Norne model. The upper plots show snapshots of the solution (after 6 and 12 years, respectively) and the lower plots show oil and water surface rates for all producers.

found in `incompExampleNorne2ph` from the `incomp` module). The only difference is how to account for the transmissibility multipliers. This is done as follows,

```

hT = computeTrans(G, rock, 'Verbose', true);
tmult = computeTranMult(G, grdecl);
hT = hT.*tmult;

```

where `grdecl` is the Eclipse input structure that contains data for the `MULTZ` keyword. The second call takes the `MULTZ` values, which are associated with cells and assigns a corresponding reduction value between 0 and 1 to all half-faces. We then multiply the half-transmissibilities `hT` by `tmult` to get the reduced transmissibility. It is important that these multipliers are assigned *before* computing the intercell transmissibilities. Altogether, approximately 6.5% of the half-faces have reduced transmissibility.

Figure 10.25 shows results of a simulation of a scenario in which water is injected into oil. The oil has a five times higher viscosity and hence the injected water will form an unstable displacement with a weak displacement front similar to what has been discussed above. The main displacement takes place in the region that involves injectors I1 to I5 and producers P1 to P3.

The last two producers are located in regions that are poorly connected to the rest of the reservoir where the injectors are placed and hence contribute less to the overall production. This is particularly true for producer P4, which most likely is completely misplaced. As we have seen in previous examples, once water breaks through in a well (primarily in P1 and P2), the oil rate decays significantly.

The main purpose of the example lies in the actual code and not in the results it produces. Discussing the simulation results beyond this point is therefore somewhat futile since the fluid system and the well pattern have limited relevance for the real reservoir, which contains a three-phase oil–gas–water system that is modelled using the compressible black-oil equations that will be discussed in the next chapter. The main take-away message is that solvers can be applied to models that have the geometrical and petrophysical complexity seen in real reservoir models.

COMPUTER EXERCISES:

71. To get more acquainted with the multiphase incompressible solvers and see their versatility, you should return to a few examples presented earlier in the book and try to set them up as multiphase test cases:
 - Consider the strange reservoir studied in Exercise [8 on page 68](#) and place one injector to the south and two producers placed symmetrically along the northern perimeter and simulate the injection of one pore volume of water into an oil.
 - Consider the test case with non-rectangular reservoir geometry in Figure [5.6 on page 168](#) and set up a simulation that injects one pore volume from the flux boundary. How would you compute the flux out of the pressure-controlled boundary?
 - Pick any of the faulted grids generated by the `simpleGrdecl` routine as shown in Figure [3.33 on page 104](#) and place an injector in one fault block and a producer in the other and simulate the injection of half a pore volume of water.
72. Consider a rectangular reservoir with two wells, e.g., as shown in Figure [3.39](#) and compare solutions computed with three different grids: a uniform coarse grid, a uniform fine grid, and a coarse grid with radial well refinement.
73. Try to study the Norne model in more detail.
 - Are the multipliers important for the simulation result?
 - Is gravity important or can it be neglected?
 - Can you come up with a better recovery strategy, i.e., improved placement and control strategy for wells?
 - Do you get very different solutions if you use a consistent solver? (Hint: Although multipliers can be incorporated into these solvers as described in [\[194\]](#), this is not part of the public implementation and for this comparison you should therefore neglect the `MULTZ` keyword).

10.4 Numerical errors

There are several errors involved in the computations above. First of all, we have the obvious numerical discretization errors that arise when approximating a continuous differential equation by a set of discrete finite-volume equations. For single-phase, incompressible flow, these errors were purely spatial errors. These errors will decrease with decreasing size of the grid as long as the spatial discretization is consistent. However, as we saw in Chapter 6, the standard two-point scheme is not consistent unless the grid is strictly K-orthogonal, and the `incompTPFA` pressure solver can in general be expected to produce errors for anisotropic permeabilities and skewed grids. When using a sequential method to solve multiphase flow equations, there will also be *temporal* errors arising from three different factors: discretization errors arising when approximating temporal derivatives in the transport equations by discrete differences, amplifications of spatial errors with time, and errors introduced by the operator splitting underlying the sequential solution procedure. In this section, we will briefly discuss the two last error types in some more detail.

10.4.1 Splitting errors

When using a sequential solution procedure, the total velocity is computed from the fluid distribution at the start of each time step. This means that the effect of mobility on the flow paths is ‘frozen’ in time, and for each time step appears as if we solved a single-phase flow problem with reduced permeability in all parts of the domain that contain more than one fluid phase. Within a single splitting step the transport solver will thus only resolve the dynamic effect of mobility along each flow path, but will not account for the fact that mobility changes reduce the effective permeability along each flow path or move the flow paths themselves. This introduces a ‘time lag’ in the simulation, which may lead to significant errors in the propagation of displacement fronts if the splitting steps are chosen too large.

Homogeneous quarter five-spot

To illustrate this, we can revisit the homogeneous quarter five-spot from Section 10.3.3 and study the self-convergence of approximate solutions defined on a fixed grid as the number of splitting steps increases. Figure 10.26 shows approximate solutions at time $t = 0.6$ PVI computed with 4^ℓ steps for $\ell = 0, \dots, 3$. With a single pressure step, the displacement front coincides with the time line from the single-phase flow field since the pressure computation only sees the initial oil saturation. The only exception is a certain smearing introduced by the spatial and temporal discretization of the explicit scheme used to compute the transport step. The retardation effect that oil has on the invading water is better accounted for as the number of splitting steps

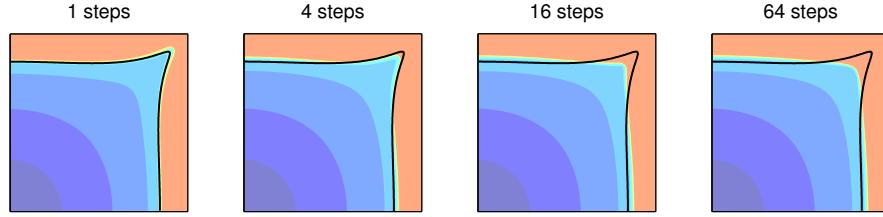


Fig. 10.26. Quarter five-spot solution at time 0.6 PVI computed on a uniform 128×128 grid with the explicit transport solver and different number of splitting steps. The solid lines are time lines at $t = 0.6a$ for a single-phase displacement.

Table 10.1. Self-convergence for the homogeneous quarter five-spot computed on a 128×128 grid with n splitting steps relative to a reference solution computed with 256 time steps after 0.6 PVI. The errors are reported in relative norms.

n	saturation		pressure	
	L ¹ -error	rate	L ² -error	rate
1	5.574e-02	—	4.950e-03	—
2	4.368e-02	0.35	5.140e-04	3.27
4	2.778e-02	0.65	1.554e-04	1.73
8	1.445e-02	0.94	4.524e-05	1.78
16	6.389e-03	1.18	1.226e-05	1.88
32	2.394e-03	1.42	2.998e-06	2.03
64	7.869e-04	1.61	5.990e-07	2.32

increases, and hence the splitting solution gradually approaches the correct solution. Table 10.1 reports the self-convergence towards a reference solution computed on the same grid with 256 splitting steps. The convergence rate is computed based on an assumption that the error scales like $\mathcal{O}(\Delta t^n)$. If the solution having error E_2 is computing using twice as many time steps as the solution having error E_1 , the corresponding convergence rate is

$$r = \log(E_1/E_2)/\log(2).$$

Pressure is smooth and will therefore converge faster than saturation, which is a discontinuous quantity and hence will have much larger errors. The convergence for high n values is exaggerated since we are measuring self-convergence towards a solution computed with the same method, but with a larger number of steps. The code necessary to run this experiment is found in `splittingErrorQ5hom.m` in the `in2p` directory of the `book` module.

Heterogeneous quarter five-spot

As pointed out earlier, the coupling between the pressure and transport equation depends on the variation in total mobility $\lambda(S)$ throughout the simulation. If the variations are small and smooth, the two equations will remain

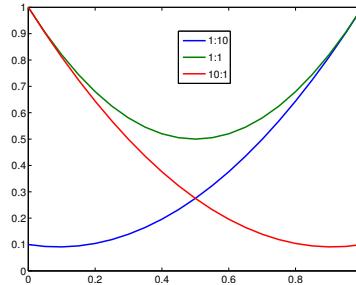


Fig. 10.27. Total mobility for three fluid models with different viscosity ratios.

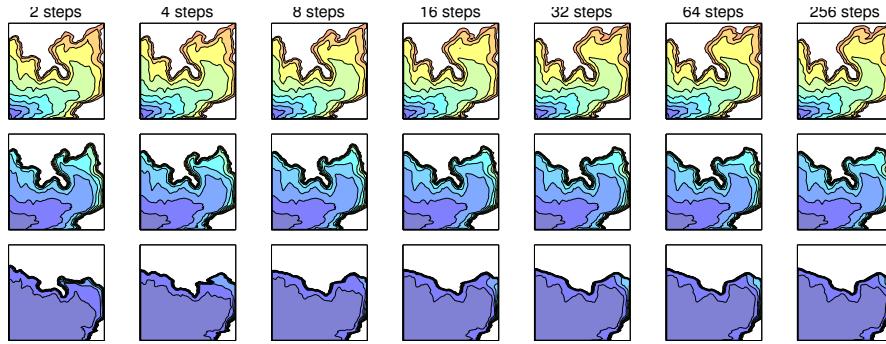


Fig. 10.28. Self convergence with an increasing number of equally-spaced splitting steps to reach time 0.5 PVI for a quarter five-spot setup on a subsample from the first layer of the SPE 10 model for three different viscosity ratios $M = \mu_w/\mu_n$ (top: $M=1/10$, middle: $M=1$, bottom: $M=10$). Saturation profiles are shown at a time scaled by $\nu(1)/\nu(M)$, where $\nu(M)$ is the characteristic wave speed of the displacement front with viscosity ratio M .

loosely coupled, and relatively large time steps can be allowed without seriously decaying solution accuracy. (For the special case of linear mobilities and equal viscosity ratios, the two equations are even completely decoupled.) On the other hand, when λ has large variations over the interval $[0, 1]$, the pressure and transport equation are more tightly coupled and we cannot expect to be able to use large splitting steps. To illustrate this, we will revisit the setup with three different fluid models used to study viscous fingering in Section 10.3.4. From the plot of the total mobilities in Figure 10.27 it is obvious that both the unfavorable (1:10) and the favorable (10:1) mobility cases will have stronger coupling between saturation and pressure than the case with equal viscosities.

Figure 10.28 shows the self convergence of the saturation profiles with respect to the number of equally-spaced time steps used to reach time 0.5 PVI. To isolate the effect of splitting errors and avoid introducing excessive numerical smearing in the solutions with few splitting steps, the transport steps have

been subdivided into substeps so that the implicit solver uses the same step length and hence introduces the same magnitude of numerical smearing in all simulations. From the figure it is clear that even with very few splitting steps, the sequential solution method manages to capture the qualitatively correct behavior for all viscosity ratios. As expected, the discrepancies in solutions with few and many time steps are larger for the favorable and unfavorable case than for the case with unit viscosity.

To investigate how the size of the splitting steps affects the *quantitative* behavior of the approximate solutions, we rerun the experiments above up to 1.5 PVI. Figure 10.29 reports water cut in the producer for all three fluid models. Starting with the unfavorable case, we see that the water cut has a dent. This is a result of the secondary finger that initially extends along the western edge making contact with the main finger and hence contributing to a more rapid incline in water production. All curves, except the one using only three time steps, follow the same basic trend. The main reason is that small variations in the saturation profile will not have a large effect on the water cut since the displacement front is so weak. With three splitting steps only, the main dent is a result of the second pressure update. For unit viscosity ratios, the production curves are still close, but here we notice a significant incline in the curve computed with 12 steps after the pressure update at 0.75 PVI. The 12-step and 48-step curves also show non-monotone behavior at 0.625 PVI and 0.69 PVI, respectively. Similar behavior can be seen for the favorable case, but since this case has an almost piston-like displacement front, the lack of monotonicity is significantly magnified. For comparison, we have also included a simulation with 96 splitting steps, in which the first two first steps have been replaced by ten smaller splitting steps that gradually ramp up to the constant time step. These profiles, and similar profiles run with 48 steps, are monotone, which suggests that the inaccuracies in the evolving saturation profiles are introduced early in the simulation when the profile is rapidly expanded by high fluid velocities in the near-well region. In our experience, using such a ramp-up is generally advisable to get more well-behaved saturation profiles. The code necessary to run this experiment is found in `splittingErrorQ5het.m` in the `in2p` directory of the `book` module.

Capillary-dominated flow

The sequential solution procedure discussed in this chapter is as a general rule reasonably well-behaved for two-phase scenarios where the fluid displacement is dominated by the hyperbolic parts of the transport equation, i.e., by viscous forces (pressure gradients) and/or gravity segregation. If the parabolic part of the solution dominates, on the other hand, a sequential solution procedure will struggle more, in particular when computing fluid equilibrium governed by a delicate balance between gravity and capillary forces. To illustrate this, we revisit the computation of capillary fringe from Section 10.3.7. If you look carefully in the accompanying code, you will see that the two cases are computed with a time step that is ten times larger for the Gaussian case than

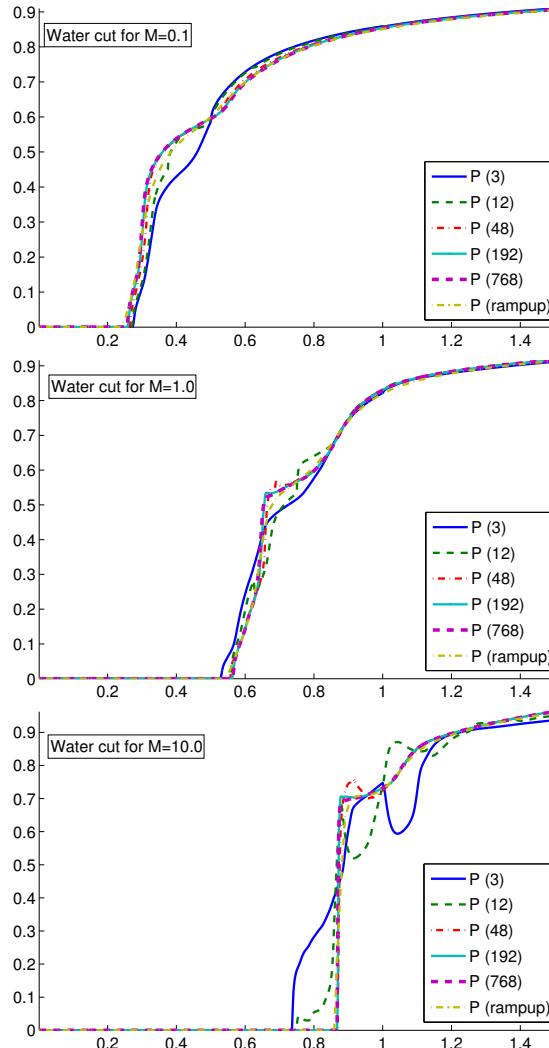


Fig. 10.29. Water cut in the producer for various number of splitting steps for the heterogeneous quarter five-spot setup from Figure 10.28.

for the case with linear permeability. The time steps (and initial the ramp-up sequence) were chosen by trial and error and are close to what appears to be the stability limit. If one, for instance, increases the final time steps by 150% for the case with linear permeability, the simulation will not converge but instead ends up predicting an oscillatory interface, as illustrated in Figure 10.30. Similar problems may arise e.g., when simulating structural trapping of CO₂ using vertical equilibrium models in which gravity gives rise to a parabolic, see e.g., [195].

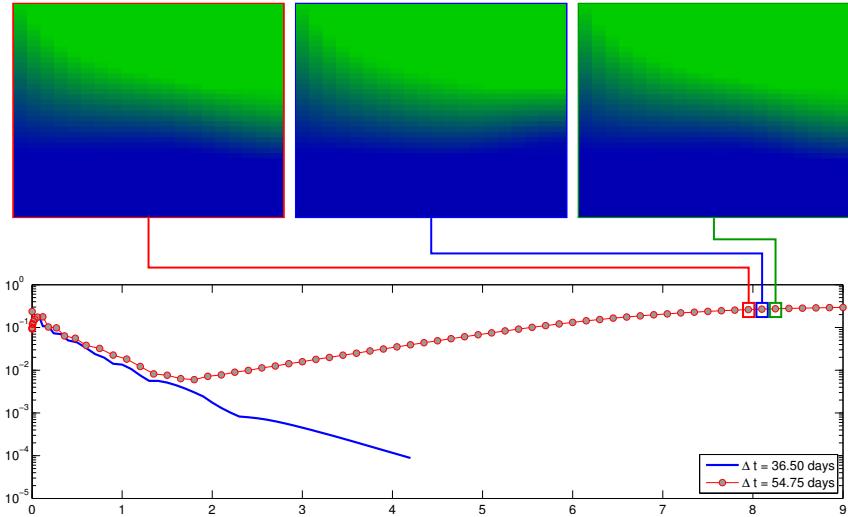


Fig. 10.30. Unstable solution computed by a sequential solution with a too large splitting step. The lower plot shows difference in L^∞ norm between consecutive time steps, which is used as convergence criterion by the solver, for a convergent solution with $\Delta t = 36.5$ days and for a divergent solution with $\Delta t = 54.75$ days. The upper plot shows three consecutive time steps for the divergent solution.

10.4.2 Grid-orientation errors

As you may recall from the discussion on page 308, the saturation-dependent mobility at the interface between two grid blocks is usually approximated by single-point, upstream mobility weighting. Like the TPFA method, the resulting scheme only accounts for information on opposite sides of a cell interface and does not take any transverse transport effects into account. It is therefore well known that this method also suffers from grid-orientation errors, especially when applied to unfavorable displacements, as we will see in the following example. In passing, we also note that to evaluate gravity and capillary-pressure terms, the transport solvers use two-point approximations similar to what is used in the TPFA method. This may introduce additional errors, but these will not be discussed in detail herein.

Homogeneous quarter five-spot

When the single-point transport solver is combined with the classical TPFA pressure solver, numerically computed displacement fronts tend to preferentially move along the axial directions of the grid, i.e., in the direction of the normal vectors of the cell faces. This will lead to grid-orientation effects like those discussed earlier in Chapter 6 even if the resulting grids are K-orthogonal. To illustrate this, we compare and contrast solutions of the standard quarter five-spot setup with a rotated setup in which the grid is

aligned with the directions between injectors and producers as illustrated in Figure 10.31. This test problem was first suggested by Todd et al. [242] and has later been used by many other authors to study grid-orientation errors in miscible displacements [263, 218, 235], which are particularly susceptible to this type of truncation error. (See `runQ5DiagPara1` for complete setup of the two cases.) To avoid introducing too much diffusion when using few time steps, we use the explicit transport solver.

For the standard setup, the combination of a single-point transport solver and two-point pressure solver overestimates the movement into the stagnant regions along the x and y axes and underestimates the diagonal movement in the high-flow direction between injector and producer along the diagonal. In the rotated setup, the grid axes follow the directions between the wells and the solvers will hence tend to overestimate the flow in the high-flow zone and underestimate the flow toward the stagnant zones. The upper row in Figure 10.32 shows that this effect is pronounced for the unfavorable displacement ($M = 0.1$), evident with equal viscosities ($M = 1$), and hardly discernible for the favorable displacement ($M = 10$). For the equal viscosity case, the difference between the two grids can be almost eliminated by increasing the number of splitting steps for a fixed Δx and/or by increasing the grid resolution provided a certain number of time steps are used. For the unfavorable displacement, the plots in the middle row of Figure 10.32 show that increasing the number of time steps changes both solutions in the same direction, but does not necessarily reduce their difference. However, since both the original and the rotated grid are regular, the grid-orientation effect introduced by the axial orientation of the grid relative to the main flow direction is reduced by increasing the grid resolution.

A possible remedy to the behavior observed above is to replace the single-point scheme by a multidimensional upwind scheme, see e.g., [129], or a modern high-resolution scheme like the ones discussed in Section 9.5. Unfortunately, such methods have not yet been implemented for general grids in MRST.

COMPUTER EXERCISES:

74. Repeat the experiments discussed above using one of the consistent solvers from Chapter 6 to compute the pressure. Do you see any differences?

Symmetric well pattern on a skew grid

We have already seen several times that since the TPFA scheme cannot approximate transverse fluxes that are parallel to grid interfaces, the `incompTPFA` solver will introduce grid-orientation errors for anisotropic permeabilities and grids that are not K-orthogonal. For single-phase flow such errors can be greatly reduced by applying the consistent solvers discussed in Chapter 6, but

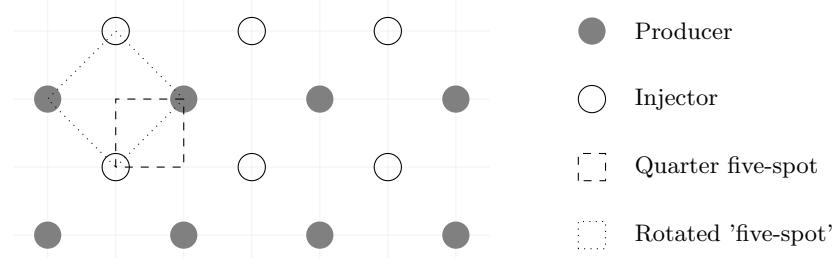


Fig. 10.31. Well setup for the quarter five-spot comparison. Displacement fronts have preferential movement parallel to the axial directions and hence the rotated setup will predict earlier breakthrough than the original setup.

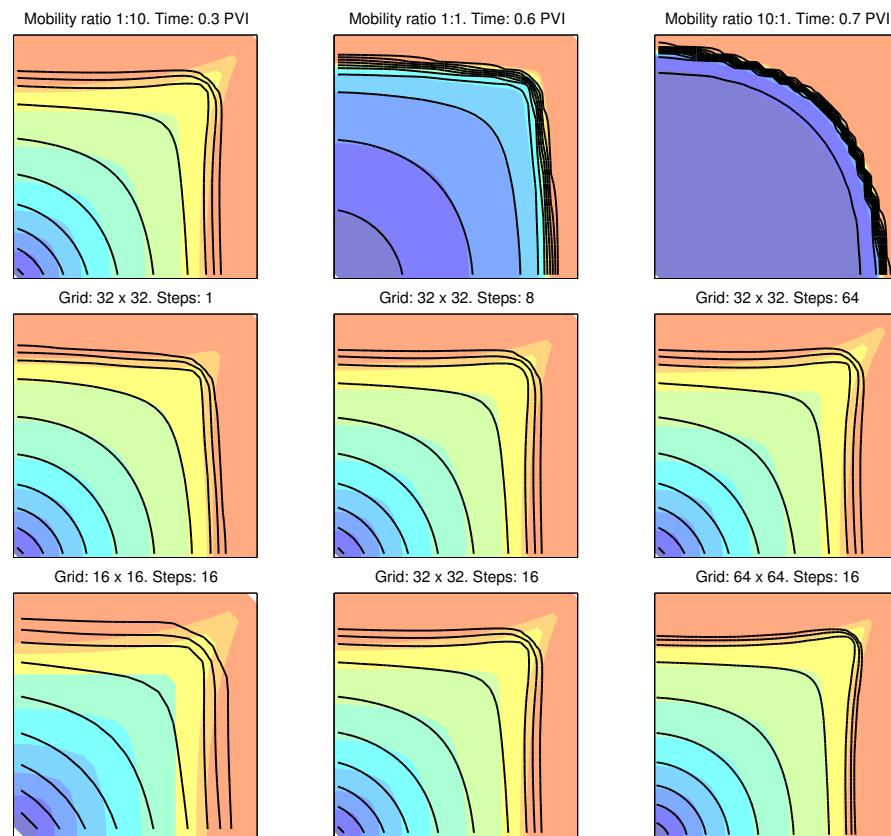


Fig. 10.32. Quarter five-spot solutions computed on the rotated (colors) and original (solid lines) geometry. The upper plots show solutions computed on a 16×16 grid with 2, 8, and 32 time steps. The lower plots show solutions computed with 16 splitting steps on a sequence of refined grids..

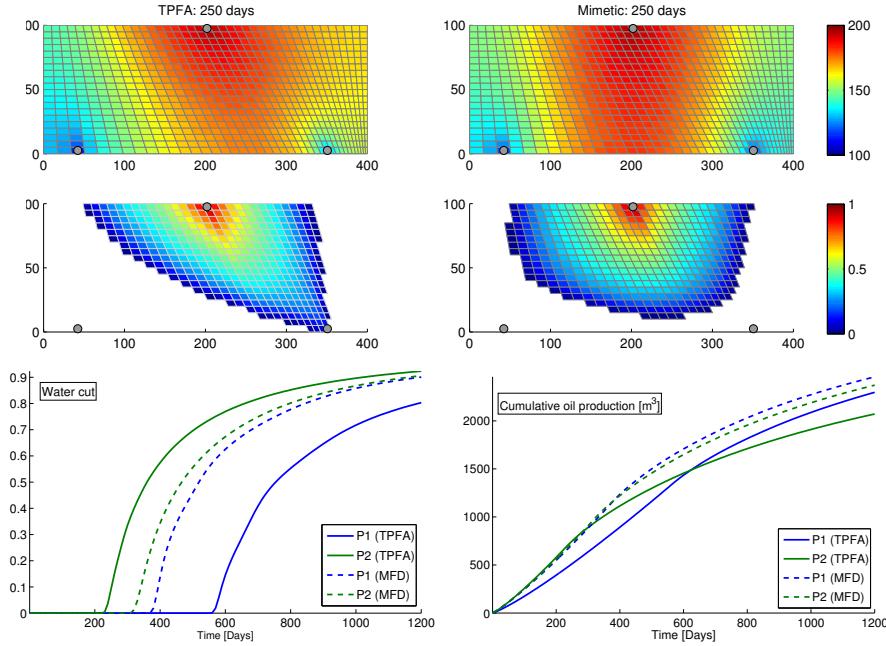


Fig. 10.33. Simulation of a symmetric flow problem in a horizontal, homogeneous domain represented on a skewed grid.

will generally not be completely eliminated. In this example, we will revisit a case discussed in Chapter 6. The computational setup consists of a horizontal $400 \times 200 \text{ m}^2$ sector from a reservoir. Water is injected from one well at the midpoint of the northern perimeter and fluids are produced from two wells along the southern perimeter, located 50 m from the southeast and southwest corners, respectively. Since the well pattern is symmetric within a confined domain and the petrophysical parameters are homogeneous and isotropic, the true displacement profile will also be symmetric. The grid, however, is skewed and compressed towards the southeast corner, and this will induce a preferential flow direction towards the southeast producer. Since grid-orientation effects are more pronounced for unstable displacements, we use the same configuration as in the previous example with a viscosity ratio $M = 10$, which gives a very mobile, weak displacement front that will tend to finger rapidly into the resident oil. Figure 10.33 shows a snapshot of the pressure and saturation profiles after 250 days along with water cuts and cumulative oil production over the whole 1200-day simulation period. The flow field computed using the `incompTPFA` pressure solver exhibits the same lack of symmetry as seen in Figure 6.5 on page 192. The result is a premature breakthrough in the southeast producer and delayed breakthrough in the southwest producer. With a mimetic finite difference (MFD) pressure solver, the water-cut curves are much closer and less affected by grid-orientation errors. Moreover, since

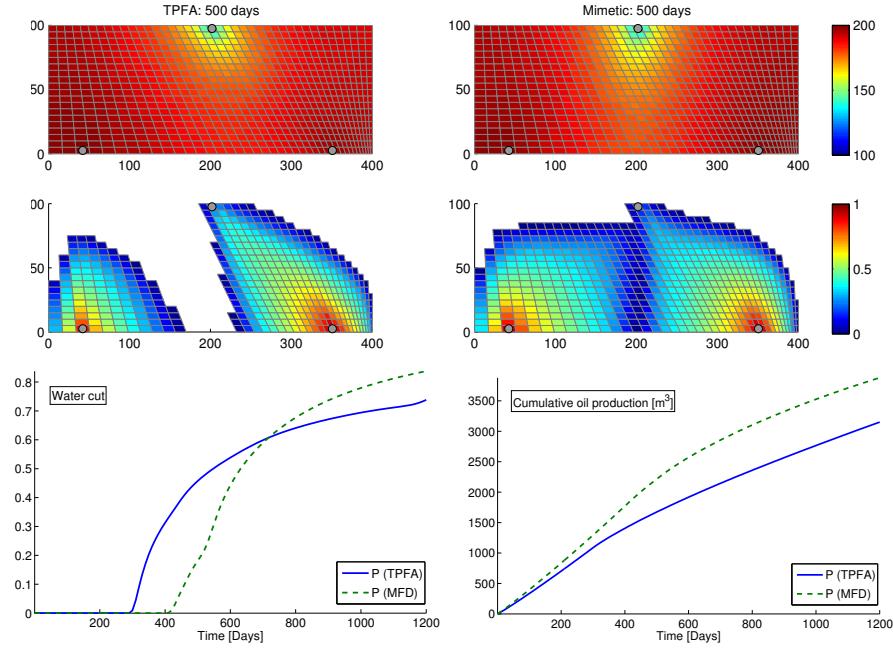


Fig. 10.34. Simulation of a symmetric flow problem in a vertical, homogeneous domain represented on a skew grid with two horizontal injectors at the bottom and a horizontal producer at the top.

all wells operate under pressure control, we see that the total oil production predicted by the TPFA scheme is significantly less than for the MFD scheme.

As a second example, we will use the same grid to describe a vertical cross-section, in which water is injected from two horizontal injectors at the bottom of the reservoir and fluids produced from a horizontal producer at the top of the reservoir. Producers and injectors operate under the same pressure-control as for the horizontal reservoir section. Figure 10.34 shows simulation results from with the TPFA and the mimetic pressure solvers. The injected and the resident fluids have a density difference of 150 kg/m^3 , and hence gravity will tend to oppose the imbibing water front in a way that accentuates the grid-orientation effects for both solvers. For comparison, Figure 10.35 reports the simulations performed on a regular Cartesian grid. Both schemes produce symmetric, but slightly different displacement profiles and the match in production profiles is largely improved compared with the skew grid.

Altogether, the two examples presented in this section hopefully show you that you not only need to take care when designing your grid, but should also be skeptic to simulations performed by a single method or a single choice of time steps. A good advice is to conduct simulations with more than one scheme, different grid types, and different time-step selection to get an idea of how numerical errors influence your results.

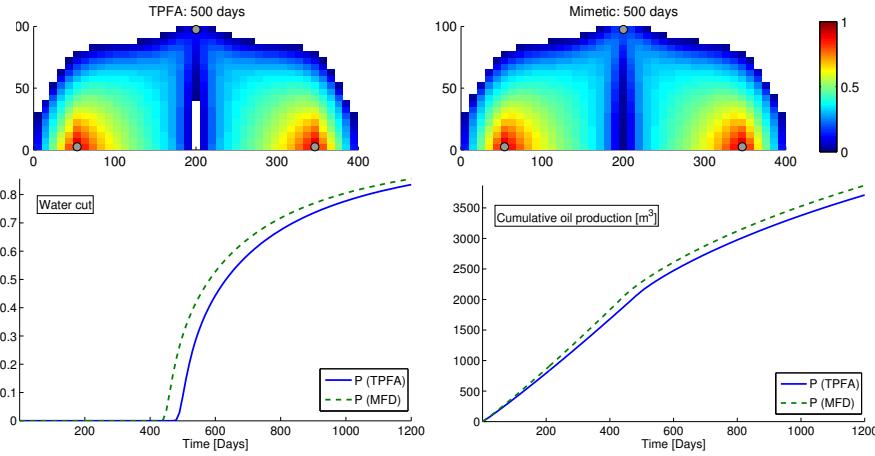


Fig. 10.35. Simulation of a symmetric flow problem in a vertical, homogeneous domain computed on a regular Cartesian grid.

COMPUTER EXERCISES:

To further investigate grid-orientation effects, consider any of the following test cases. Do you see any difference between using TPFA and a consistent pressure solver (mimetic or MPFA-O)?

75. Set up a flow problem to compare solutions computed on the extruded Delaunay or Voronoi grids shown in Figure 3.34 on page 104.
76. Set up a quarter five-spot test using the grids shown in Figure 3.46 on page 115.
77. The CaseB4 test case shown in Figure 3.20 on page 86 is represented in two different ways, using either a deviated pillar grid or a stair-stepped grid, and sampled at two different resolutions. Set up a flow problem with four wells, two injectors and two producers, one in each corner of the reservoir. Run simulations on all four grids and compare production curves. You can either use homogeneous permeability or a layered permeability with homogeneous properties within each layer.
78. Pick any of the models in the `bedModels1` or `bedModel2` data sets and run two-phase simulations injecting one pore volume from south to north and from west to east.

11

Compressible Multiphase Flow

The black-oil equations, which were briefly introduced in Section 8.2.3, is the industry-standard approach to describe compressible three-phase flow. The purpose of this chapter is to introduce you to the underlying physics and describe the various rock-fluid and PVT properties that enter these models. We will also explain the basics of how these models are implemented in MRST.

Black-oil models generally have stronger coupling between fluid pressure and transport of phases and components than the two-phase, incompressible flow models discussed in the previous chapter. For this reason it is common to use a fully coupled solution strategy, in which the whole system of equations is discretized implicitly and all primary unknowns are solved for simultaneously. As a gentle introduction to the numerics of the black-oil equations, we pick up again the discrete operators and simple AD solvers from Chapter 7 and show how to apply them to develop a compact simulator for compressible two-phase flow without mass transfer between the phases. This code will be applicable to general unstructured grids in 2D and 3D, but it is still a far stretch from a full-fledged reservoir simulator capable of simulating real assets.

A commercial reservoir simulator is a complex software that involves a wide variety of flow models, computational algorithms, and a lot of subtle tricks to ensure robust behavior. Covering all of these features, as well as the many different scenarios a simulator can be used for, is well beyond the scope of this chapter. Instead, I will try to provide you with a minimum insight of the key principles and show how to set up your first black-oil simulation in MRST using the AD-OO framework, which will be introduced in more detail in Chapter 12. However, before we dive into models and implementation, I will try to briefly outline the full complexity of reservoir simulation and point out important topics that are not discussed in this book.

Flow models and phase behavior

Reservoir fluids are complex mixtures of a large number of chemical component that will distribute differently among the fluid phases present in the

reservoir, depending upon temperature and pressure. The composition and characteristics of hydrocarbon fluids vary largely from reservoir to reservoir, but may also have large variations across reservoir compartments. As we have seen already, the black-oil model groups the chemical components into two pseudo-components reflecting the phase they belong to at surface conditions. At reservoir conditions, each component can be present in both a liquid oleic and a gaseous phase. The phase diagrams used to describe how the phase distribution of the pseudo-components will be motivated and briefly explained, but a more detailed discussion of the underlying physics and the many different forms these diagrams can take is outside the scope of this chapter.

The black-oil equations can also model other processes than hydrocarbon recovery. Enhanced oil recovery – in which various forms of chemical and biological substances are added to the injected fluid to increase sweep efficiency and improve local displacement – can to a large degree be modelled by the same type of flow models, by adding extra fluid components to the three phases. MRST offers an EOR module that implements basic models for polymer and surfactant flow; the interested reader should consult [26] for more details. In geological storage of CO₂, the buoyant and supercritical phase formed when CO₂ is injected sufficiently deep into the subsurface will migrate rapidly towards the top of the formation, but also dissolve into the resident brine. The behavior of the CO₂–brine system can thus be modelled as a hydrocarbon system in which brine plays the role as oil and CO₂ the role as gas. Extensive support has been developed in MRST to describe the long-term migration of buoyant CO₂ in large-scale aquifer systems, in which fully implicit solution methods developed for the black-oil equations play a key role. The interested reader can find a comprehensive discussion in the thesis by Andersen [15].

Last but not least, grouping the hydrocarbons into two pseudo-components will not describe all recovery process with sufficient accuracy. In compositional simulation, the reservoir fluids are modeled as a system with M components, for which phase behavior is described by an equation of state. The basic conservation equations for such systems is given in Section 8.2.2 and typical examples of equations of state were briefly reviewed as part of our discussion of single-phase flow in Section 4.2. As of release 2017b and onward, MRST offers a module for compositional simulation. This will not be discussed herein at all; preliminary details can be found in the thesis by Møyner [176].

Wells

In previous chapters, wells have always been completed in a single cell or along straight lines following the axial directions, and have either been controlled by constant pressure or constant fluid rate. Real wells are significantly more complex and will generally follow highly deviated trajectories and may also consist of multiple branches. Many subtleties go into well modeling. In particular, special care is needed in the solution algorithm to handle cross-flow, in which fluids may flow into one part of the wellbore and out of another.

To ensure that each well operates under realistic engineering constraints, reservoir simulators offer a comprehensive set of control options that can be applied to fluid rates and bottom-hole or tubing-head pressures. A typical control strategy consists of setting a target value on a specific rate or pressure, and constraints on any number of the other quantities. This means that the well control may switch depending upon the computed pressures and flow rates once the well fails to meet one of its constraints. To reproduce production history as closely as possible, well controls can be imposed in so-called history mode so that observed fluid rates are set as input. Last, but not least, controls can also include economic constraints so that wells may also be completely shut in (or re-completed), if their production fall below a lower economic limit. The control strategy may also involve gas lift or other mechanisms introduced to help lift the well fluids to the surface when the oil or liquid rate falls below a specified limit. (This will strongly affect the bottom-hole pressure in individual wells.) Control strategies are not necessarily applied to a single well at the time, but can be imposed on groups of wells, or on the field as a whole. Typical strategies here either involve guide controls, in which total rates are distributed to individual rates according to their (production) potential, or prioritization scheme, in which wells are turned on in decreasing order of priority. Group and field controls may also involve various forms of restrictions in terms of total pump and lift capacities, re-injection of produced formation water, limitations in various surface facilities, etc. Altogether, the resulting well-control strategies may involve a lot of quite intricate logic, and only a limited number of options have so far been implemented in MRST. I therefore consider a more detailed discussion of control strategies to be outside the scope of this book.

Many wells also have advanced completions and devices for inflow control which can be modelled using *multisegment* well models. In these models, the well trajectory is discretized into a series of segments that are linked together into branches, which, in turn, may contain sub-branches. Each segment describes how the flow rate between two points (nodes) relates to the pressure drop between the same two points. The two nodes can be located at different positions along the wellbore, or on opposite sides of devices that restrict the flow, such as chokes and valves. Each node describes mass conservation, and can thus be used to describe wellbore storage effects. As of release 2017a, MRST contains a general multisegment model than allows branches of segments to be defined in both the tubing and the void space (annulus) between the inner and outer tubing. The branches can be joined at multiple points in a flexible manner to form a general network. Altogether, this enables accurate description of inflow devices and complex cross-flow patterns that may arise in multilateral wells. For brevity, a detailed discussion of multisegment wells is considered outside the scope of this book. However, as is always the case with MRST, the interested user is free to consult the code to understand the necessary details.

Data input

To model real assets, a simulator needs comprehensive and flexible input functionality to specify the setup of all ingredients that constitute the simulation model. The industry-standard approach is to define an input language consisting of a number of keywords that can be used to construct a model of a computation (i.e., a state machine). The detailed syntax will vary from one simulator to another, but the general setup is conceptually the same. You have already encountered the ECLIPSE input format [226, 29] multiple times throughout the book. Over the years this software has amassed options for simulating a wide variety of physical processes and the input language currently has several thousand keywords, many of which have multiple available options. There are also similar file formats that can be used to store simulation results so that these can be reloaded and used to continue a previous simulation.

So far, we have only discussed how ECLIPSE input describes grids and petrophysical properties, and even for this purpose, there are a large number of keywords we have not discussed. In general, an ECLIPSE input deck consists of eight different sections, some of which are optional:

- RUNSPEC – required section, includes a description of the simulation, including parameters such as name of the case, grid dimensions, phases and components present, number of wells, etc.
- GRID – required section, describes basic grid geometry/topology and petrophysical properties (porosity, permeability, net-to-gross).
- EDIT – optional section describing user-defined changes that are applied to the grid data *after* these have been processed to modify pore volume, cell centers, transmissibilities, local grid refinements, etc.
- PROPS – required section describing fluid and rock properties such as relative permeabilities, capillary pressure, and PVT specification as function of fluid pressures, saturations and compositions.
- REGIONS – optional section defining how the computational grid can be split into multiple regions for calculating PVD properties, relative permeabilities and capillary pressures, initial conditions, and fluids in place.
- SOLUTION – required section specifying how the model is to be initialized: calculated from specified fluid contact depths, read from a restart file from a previous run, or specified for each grid cell.
- SUMMARY – optional section that enables the user to specify the reservoir responses (well curves, field-average pressure, etc) to be written to a summary file after each time step.
- SCHEDULE – required section, defines the wells and how they are to be operated, specifies time step selection and various tolerances for the solvers, controls the output of quantities defined over the grid (pressures, saturations, fluxes, etc).

The sections *must come in this prescribed order* in the input file, but the keywords within each section can be given in arbitrary order. Altogether, the

input data file can be seen as a kind of scripting language that not only enables the user to input data to the simulator, but also provides control of the time dependency and how the simulator will run a prescribed set of time steps. In the following, we will borrow the name convention from ECLIPSE and refer to the way a simulation is to be run as a *schedule*. In the simplest case, a schedule consists of a number of prescribed time steps and one control per well. However, schedules representing real assets tend to be much more complicated, as can be inferred from the discussion of wells above. Let me also make two additional remarks: Since several of the keywords, and in particular those in the GRID section, may contain huge amounts of data, the input format offers an INCLUDE keyword allowing input data to be separated into multiple files. Moreover, although MRST aims to support the most widely used keywords in the required sections, there are many keywords that are *not* supported. In particular, no effort has been put into supporting the flexible, yet complex, options for user-specified postprocessing of geological models in the EDIT section.

In the following, I will only go into details about ECLIPSE input [226, 29] when this is needed to understand how MRST works. For readers interested in a more thorough introduction, I recommend the lecture notes by Pettersen [213], which are much to the point, freely available, and complementary to the discussion in my book. An early version of the ECLIPSE 100 User Course [228] can also be found online.

Numerical methods

As should be evident from the above discussion, the discrete flow models representing compressible, three-phase flow are significantly more complex than the incompressible, two-phase flow equations considered in the previous chapter. For small models with relatively simple phase behavior and solution trajectories, it is straightforward to linearize the flow equations and solve them with standard linear algebra, much in the same way as we did for single-phase flow in Chapter 7.2. However, for more complex phase behavior, there are many features complicating the solution process. Since all phases may not always be present, one needs robust strategies for switching primary unknowns from one cell to another. Likewise, extra post-processing steps may be needed to record historic solution trajectories to model hysteretic behavior of relative permeabilities and capillary pressure, and so on. As the complexity increases for each of the three constituents of the reservoir model (geology model, fluid model, well model), one really sees the benefit of using automatic differentiation to perform the necessary linearizations.

All these model complications generally translate to increased nonlinearity in the discretized system, and a simple nonlinear Newton–Raphson solver cannot be expected to converge unless we instrument it with inexact line-search methods and methods for time-step control as briefly introduced in Section 10.2.2. Likewise, there will be a need for correct scaling of the residual equations as well as more advanced linear solvers, e.g., a preconditioned

Krylov subspace method like the flexible general minimal residual (GMRES) method, combined with specialized preconditioning strategies aimed at lessening the impact of the largely different mathematical characteristics of the various sub-equations. Classic choices of single-stage preconditioners include the incomplete lower/upper (ILU) family or the nested-factorization method [20], which is still the default method in ECLIPSE. Another popular choice is the two-stage constrained-pressure residual (CPR) method [253]. Here, the first step computes an approximate solution of a reduced-pressure matrix that is appropriately scaled and selected so that one can use, e.g., a very efficient algebraic multigrid solver [238, 95]. The second step then applies a broadband smoother like ILU to the entire system. More details about preconditioners and the exact solution strategy used in MRST to obtain robust simulation of real cases will not be presented herein. The interested reader can, as always, find all details in the code.

Details of the MRST implementation

To a large degree, MRST has many of the model complexities discussed above implemented in its black-oil type solvers. Our first successful attempt at implementing black-oil simulators relied on discrete operators for discretization and automatic differentiation and used the type of procedural programming that has been discussed so far in the book. As more complex behavior was introduced, the code gradually became more unwieldy, and at a certain stage, we realized that a major restructuring was required to make the code easier to extend and maintain. The result was the object-oriented AD-OO framework [139, 176, 26], which will be discussed in more detail in the next chapter. The key idea of the AD-OO framework is to separate the implementation of physical models (i.e., conservation equations and constitutive relationships), discrete operators, nonlinear solvers and time-stepping, and assembly and solution of the linear system. By using a combination of classes, structures, and functions for generic access, details from within each of these contexts can be hidden from the user/developer and only exposed if really needed. As a result, it is now much simpler to implement generic algorithmic components or extend existing models with more comprehensive flow physics.

This introduces a pronounced divide in MRST between simulators written for incompressible and compressible flow. It also represents a pedagogical challenge for this book, which I have solved by relying on the simple codes written in a procedural form to introduce the key concepts that are also found in the object-oriented framework, albeit implemented in a somewhat different form. Once the basic concepts have been introduced, I switch to using the AD-OO framework when presenting simulation cases with black-oil models. This means, in particular, that the codes used to run black-oil simulations will not be presented at the same level of detail as in previous chapters of the book, but full details can as usual be found by consulting the source code.

11.1 Two-phase flow without phase transfer

For a system consisting of two immiscible phases, each with a fixed chemical composition and no mass transfer between the phases, the basic flow model consists of the conservation equations (8.21) and the standard multiphase extension of Darcy's law, (8.22). This is often referred to as a *dead-oil* model, and is representative for reservoirs with relatively thick oil or residue that has lost its light and volatile hydrocarbon components, or for oils containing a constant amount of dissolved oil because pressure is so high that gas cannot appear as a separate phase (more about this later).

If we for simplicity neglect capillary pressure and introduce discrete approximations for all derivatives, the discrete flow equations read,

$$\frac{(\phi S_\alpha \rho_\alpha)^{n+1} - (\phi S_\alpha \rho_\alpha)^n}{\Delta t^n} + \operatorname{div}(\rho v)_\alpha^{n+1} = (\rho q)_\alpha^{n+1}, \quad (11.1a)$$

$$v_\alpha^{n+1} = -\frac{K k_{r\alpha}}{\mu_\alpha^{n+1}} [\operatorname{grad}(p^{n+1}) - g \rho_\alpha^{n+1} \operatorname{grad}(z)]. \quad (11.1b)$$

There are only two issues that are conceptually different from the corresponding single-phase model (7.2) studied in Chapter 7: First of all, we now have two phases and must ensure that all saturation-dependent quantities are averaged correctly, as discussed in the previous chapter. Secondly, wells must be treated slightly different, since the wellbore now may contain a mixture of two fluid phases. We put this issue off now, and return to it later.

Our first two-phase simulator

To develop our first simulator, we consider a simple rectangular $200 \times 200 \times 50$ m³ model, which is initially filled with oil with oil in the upper 2/3 and by water in the bottom 1/3 of the volume. Water is injected in the lower southwest corner and fluids are produced from the upper northeast corner. To mimic the effect of injector and producer wells, we manipulate the flow equations to ensure reasonable inflow/outflow for the perforated cells. The resulting code called `twoPhaseAD.m` can be found in the `book` module and follows essentially the same steps as the single-phase code in Section 7.2. The setup of the geological model is essentially the same and we hence skip the details. For the fluid model, we assume that the water phase is slightly compressible with quadratic relative permeabilities, whereas the oil phase is lighter, more compressible, and has cubic relative permeabilities

```
[muW, muO] = deal(1*centi*poise, 5*centi*poise);
[cw, co] = deal(2e-6/psia, 1e-5/psia);
[krW, krO] = deal(@(S) S.^2, @(S) S.^3);
```

The reference densities at 200 bar are 1014 kg/m³ for water and 850 kg/m³ for oil. Since the capillary pressure is assumed to be zero, the initial pressure distribution can be computed exactly as for single-phase flow on page 209. The transmissibilities and the discrete operators are independent of the flow

equation and are defined exactly as in Section 7.2.2, except that we now also have to define the upwind operator:

```
upw = @(flag, x) flag.*x(N(:, 1)) + ~flag.*x(N(:, 2));
```

The setup of the simulation loop follows along the same path as in Section 7.2.4. We start by initializing the unknown variables and indices that later can be used to extract the from the global vector of unknowns,

```
[p, sW] = initVariablesADI(p0, sW0);
[pIx, sIx] = deal(1:nc, (nc+1):(2*nc));
```

We start by initializing the necessary iteration variables and creating a simulation schedule containing a gradual ramp-up of the time step, as discussed in the previous chapter. Then, the main loop is more or less identical to the one shown on page 213. The only difference is how we compute the discrete equations. We start by computing all cell-based properties

```
% Densities and pore volumes
[rW,rW0,r0,r00] = deal(rhoW(p), rhoW(p0), rho0(p), rho0(p0));
[vol, vol0] = deal(pv(p), pv(p0));

% Mobility: Relative permeability over constant viscosity
mobW = krW(sW)./muW;
mob0 = kr0(1-sW)./mu0;
```

In the flux terms, the product of density and mobility at the interface between cells is approximated using upwind-mobility weighting. To this end, we first define the difference in phase pressures across each cell interface,

```
dp = grad(p);
dpW = dp-g*avg(rW).*gradz;
dp0 = dp-g*avg(r0).*gradz;
```

which we then use to determine the upstream direction for each phase and pick cell value from the upstream side

```
vW = -upw(double(dpW) <= 0, rW.*mobW).*T.*dpW;
v0 = -upw(double(dp0) <= 0, r0.*mob0).*T.*dp0;
```

Now, we have all terms that are needed to compute the residual equations in each cell,

```
water = (1/dt(n)).*(vol.*rW.*sW - vol0.*rW0.*sW0) + div(vW);
oil = (1/dt(n)).*(vol.*r0.(1-sW) - vol0.*r00.(1-sW0)) + div(v0);
```

The injector is only set to inject 0.1 pore volume of water over a period of 365 days. We model the injection by a source term having constant mass rate equal the total volumetric rate times the surface density of water

```
water(inIx) = water(inIx) - inRate.*rhoWS;
```

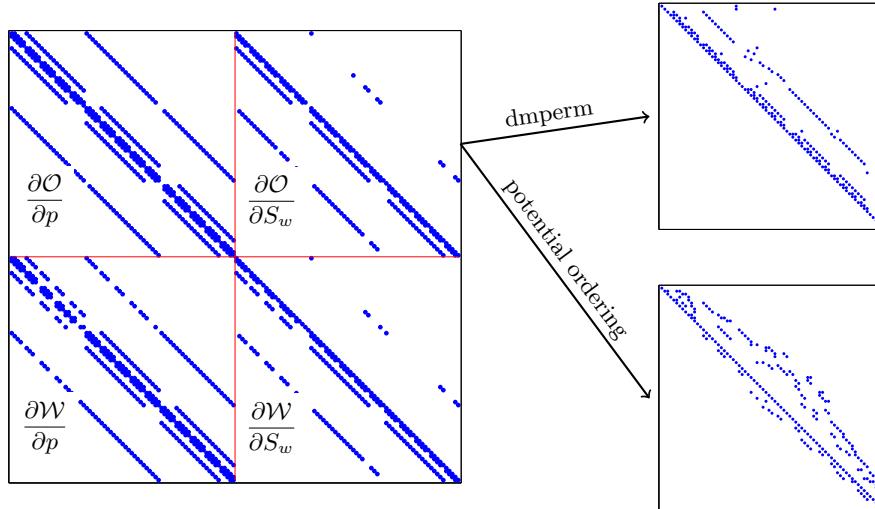


Fig. 11.1. Linearized equations for the compressible, two-phase system without mass transfer between phases. The discrete equations have a 2×2 block structure corresponding to the derivatives of the water (W) and oil (\mathcal{O}) equations with respect to the two primary variables p and S_w .

For the producer, we just fake plausible behavior; that is, we replace the oil equation by an equation that equates the cell pressure to the specified bottom-hole pressure and set the residual equation for water equal to S_w ,

```
oil(outIx) = p(outIx) - outPres;
water(outIx) = sW(outIx);
```

These conditions will ensure that fluids flow into the perforated cell.

Structure of the linearized system

As before, the linearized system can be assembled into a global system matrix by concatenating the two ADI vectors,

```
eqs = {oil, water}; eq = cat(eqs{:});
```

Let us inspect this system in some detail. To make the structure of the matrix as transparent as possible, we use consider a small $5 \times 5 \times 3$ grid. In the first iteration steps, a large number of the coefficients in the residual equations will be zero since the pressure differential induced by the wells has not yet started to cause significant movement of fluids. We therefore run five of the ramp-up steps, bringing us to time 5 days, before we inspect the linearized system Figure 11.1 shows the resulting linearized system. Here, we recognize the standard 7-point scheme for an elliptic equation for the matrix blocks corresponding to derivatives with respect to p . For the derivatives respect to

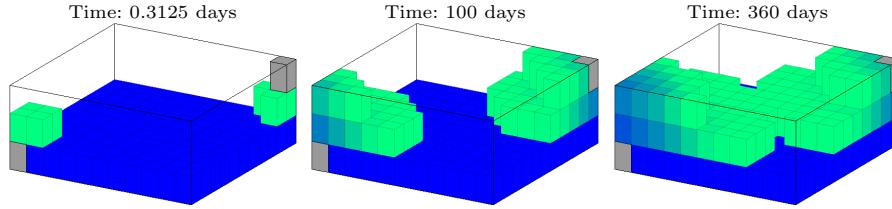


Fig. 11.2. Saturation profiles for the simple two-phase problem without mass transfer between the phases. The plots only show cells in which $S_w > 10^{-3}$.

S_w , we can use a reordering of the block to recognize the triangular structure we discussed in Section 10.3.4, which is typical for a hyperbolic problem. Indeed, in the absence of capillary forces, we have already seen that this compressible system can be rewritten as a parabolic pressure equation and a hyperbolic transport equation.

Figure 11.2 clearly shows that our suggested production strategy might not be optimal. Already in the first time step, water is sucked up from the bottom layer and towards the producer. (Maybe you can think of a better design to produce this reservoir?) However, by injecting in the water layer, rather than in the oil zone, we ensure that oil is not pushed down into the water zone. (Try to move the injector to the upper southwest corner and repeat the simulation.)

Comparison with incompressible case

Next, let us investigate what effect compressibility has on the system by comparing solutions in the compressible and the incompressible case. However, if we reset all the compressibilities to zero and rerun the code, we get the following error message:

```
Warning: Matrix is close to singular or badly scaled.  
Results may be inaccurate.  
RCOND = 2.977850e-23.
```

This is usually reason to worry, and we will get back to this issue shortly. However, let us first look at the computed results in terms of the oil production, the pressure in the cell perforated by the injector, and the field-average pressure. The oil production can be measured from a simple mass-balance calculation. To this end, we need the initial oil in place, which we compute as part of the initialization.

```
ioip = sum(pv(p0).* (1-sW0).*rho0(p0));
```

The injector pressure, average field pressure, and oil in place at the end of each time step can be computed by preprocessing the array of structures we used to store the computed solutions,

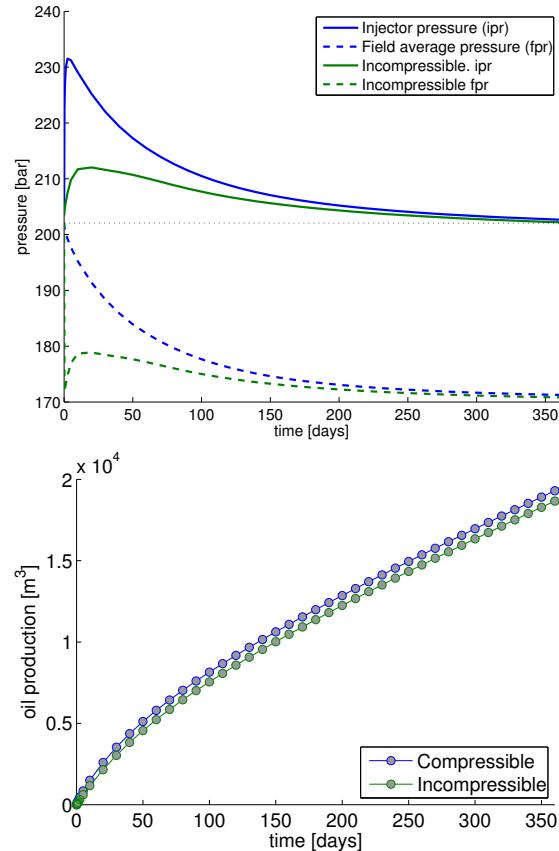


Fig. 11.3. Reservoir responses for the first two-phase example without mass transfer. The upper plot shows the pressure in the perforated cell and field-average pressure (FPR) and the lower plot shows total field oil production (FOPT). (The dotted line indicates initial FPR).

```
t = arrayfun(@(x) x.time, sol)/day;
ipr = arrayfun(@(x) x.pressure(inIx), sol)/barsa;
fpr = arrayfun(@(x) sum(x.pressure)/G.cells.num, sol)/barsa;
oip = arrayfun(@(x) sum(pv(x.pressure).*((1-x.s).*rho0(x.pressure))), sol);
```

Notice here the combination of the `arrayfun` function, which enables us to operate on each element of an array, and anonymous function used to extract and process the fields belonging to each array element. Figure 11.3 reports these reservoir responses. Both the pressure in the perforated injector cell and the field-average pressure shown in the upper plot, exhibit significantly different behavior in the compressible and incompressible models. In the compressible

case, we see that the injector pressure¹ is significantly higher since some of the injected energy will be used to compress the oil and expand the pore space, rather than only pushing fluids, as in the incompressible case. At the same time, the reduced pressure at the producer will cause the compressible oil to expand. Hence, we also observe a somewhat higher total field-oil production. Looking at the field average pressure, we see that this is high initially in the compressible case, but decays gradually as more fluids are produced. The same trend is also observed in the incompressible case, but here we have an initial transient period during which the pressure first decreases and then increases slightly, before we observe the expected gradual decay. Given that we do not have a real well model, it is somewhat futile to put too much attempt into interpreting this behavior.

Numerical challenges

Let us return to the numerical difficulties of the problem. Figure 11.4 reports the number of iterations observed as well as a plot of how the reciprocal condition number changes with time for the two simulations. The more well-conditioned the system is, the closer the reciprocal condition number should be to unity. All condition numbers are very small in our case, indicating quite ill-conditioned systems. In particular, during the first four iterations steps, the reciprocal condition numbers for the incompressible case are of the order 10^{-37} to 10^{-24} , which means that this system is effectively singular. Nevertheless, the direct solver manages to compute a solution that can be used to continue the nonlinear iteration toward convergence. This is a well-known problem in reservoir simulation; fully implicit formulations are not suited for solving incompressible flow problems. To avoid singular matrices, it is therefore a common trick to add a little compressibility to the problem to make it less ill-posed. In our example, we could, for instance, assume incompressible fluids but set the rock compressibility to 10% of the value used above without visible effects on the pressure and production curves reported in Figure 11.3.

Also in the compressible case, the condition numbers are not very good. From a pragmatic point of view, you may think that this is not a serious problem since the direct solver in MATLAB clearly has no problem to solve system. Nevertheless, the problem will get worse when the system has strong heterogeneity. Moreover, when moving to larger system, we would eventually have to utilize an iterative solver, which would most likely need a suitable preconditioner to deliver satisfactory convergence as we will see later.

To summarize: Through this simple example, you have been introduced to the typical structure of a compressible, multiphase problem and seen that the elliptic and hyperbolic characteristics discussed in previous chapters indeed can be observed in the discrete system. We have also demonstrated that

¹ Referring back to the Peaceman well model (4.34) on page 132, this pressure corresponds to p_R and will be lower than the bottom-hole pressure p_{bh} inside the wellbore.

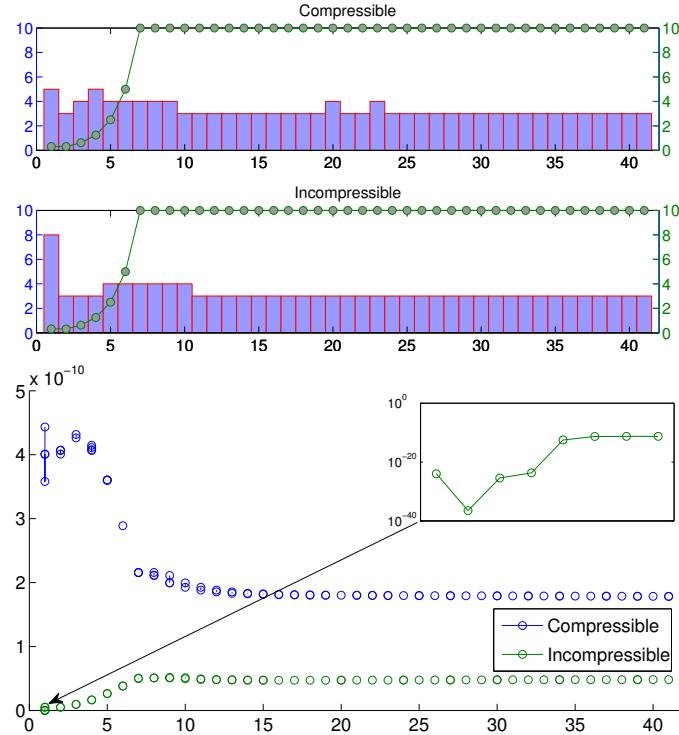


Fig. 11.4. Numerical stability for the two-phase problem. In the upper plot, the bars show the number of iterations to convergence for each of the 41 time step; time-step lengths (in units days) are shown as a line plot. The lower plot shows estimates of the variation in reciprocal condition numbers for the linearized system.

there are significant differences in compressible and incompressible systems and that these generally require different solution strategies; hence the divide in MRST between incompressible and compressible solvers. We will return to more numerical examples later in the chapter, but now it is time for a more in-depth discussion of the fluid and phase behavior of three-phase models.

11.2 Three-phase relative permeabilities

When extending our two-phase simulator to three phases, we have to determine how to model saturation-dependent quantities like capillary pressure and relative permeabilities. In principle, one could imagine that all three phases could flow simultaneously in all parts of the pore volume and that relative permeability and capillary pressure generally were functions of two independent saturations. In industry, it is more common to assume that the flow is essentially two-phase, with the third phase acting as a passive back-

ground phase that restricts the flow volume and provides resistance to flow but does not contribute actively to the flow. Effectively, this means that gas and water relative permeabilities are functions of gas and water saturation, respectively, as we saw in Section 8.1.4. (The same applies to capillary pressure). A commercial simulator like ECLIPSE, offers several different options [227], including both the Stone I and II models. Common for all options is that they are based on interpolation of tabulated input. The various options are classified into families that either use one-dimensional or two-dimensional input. This offers great flexibility, but may also slow down the simulation if the user is not careful when specifying the input tables. Fully implicit simulators implemented with the AD-OO framework in the `ad-blackoil` module utilize fluid objects as discussed in Sections 5.1.1 and 10.1 to evaluate relative permeabilities and capillary pressures as general functions of phase saturations (and component concentrations). It is up to the user whether these functions should be based on analytic expressions or tabulated input, and the default dependence on saturation only can easily be overridden should you need to include dependence on other primary variables.

For convenience, the `ad-props` module offers a generic interface for constructing fluid objects based on input in ECLIPSE format. So far, we have only implemented the most common family of input parameters and the default option for interpolating three-phase oil relative permeability from two-phase curves. Nonetheless, it should not be a very difficult exercise to extend this implementation to include other families of input options. In the rest of this section, I will describe the default model and its implementation in the `ad-props` module in more detail.

Default relative permeability model from ECLIPSE 100

The default model for calculating three-phase oil relative permeability in ECLIPSE 100 is based on the same key idea as the Stone I model (8.19)–(8.20) on page 247, but where the Stone I model assumes a nonlinear relationship to interpolate the two-phase curves, the ECLIPSE 100 model employs a simpler linear relationship, which according to [227] avoids problems of negative values and poor conditioning. The model makes the key assumption that water and gas are completely segregated within each cell, giving the fluid distribution illustrated in Figure 11.5. This means that each cell is divided into two zones (here, S_w , S_o , and S_g denote the average saturations):

- in the upper gas zone, the oil saturation is S_o , the water saturation is S_{wc} , and the gas saturation is $S_g + S_w - S_{wc}$;
- in the lower water zone, the oil saturation is S_o , the water saturation is $S_g + S_w$, and the gas saturation is 0.

Then, the oil relative permeability is interpolated linearly from two-phase relative permeabilities as follows

$$k_{ro}(S_g, S_w) = \frac{S_g k_{rog}(S_o)}{S_g + S_w - S_{wc}} + \frac{(S_w - S_{wc}) k_{row}(S_o)}{S_g + S_w - S_{wc}}, \quad (11.2)$$

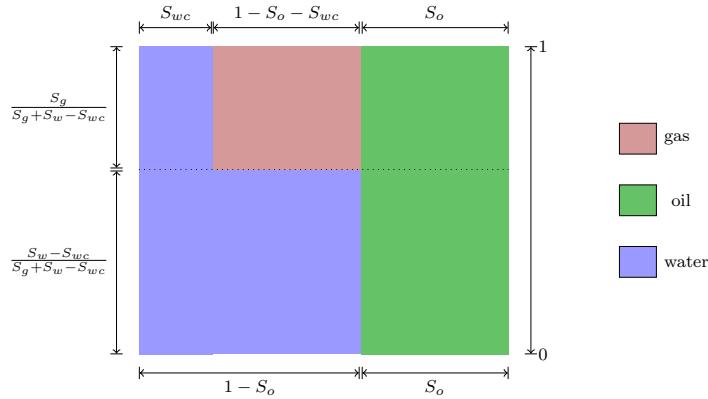


Fig. 11.5. Fluid distribution inside each cell assumed by the default method for computing three-phase oil relative permeability in MRST; adapted from [227].

with $S_o = 1 - S_w - S_g$. Here, k_{rog} is the oil relative permeability for a system with oil, gas, and connate water, and k_{row} is the oil relative permeability for a system with oil and gas only, both tabulated as a function of oil saturation.

Relative permeabilities in the ad-props module

The relative permeability model described above is implemented inside the `ad-props` module in the function `assignRelPerm`. Before this function is called, appropriate routines must have been run to assign data tables and functions for interpolating these data to the fluid object `f`. We will return to how this is done shortly. The `assignRelPerm` function inspects the fields that are present in the fluid object and assigns the correct relative-permeability functions accordingly

```
function f = assignRelPerm(f)
    if ~isfield(f, 'krOG')      % two-phase water/oil
        f.relPerm = @(sw, varargin)relPermWO(sw, f, varargin{:});
    elseif ~isfield(f, 'krOW')  % two-phase oil/gas
        f.relPerm = @(sg, varargin)relPermOG(sg, f, varargin{:});
    else
        % three-phase
        f.relPerm = @(sw, sg, varargin)relPermWOG(sw, sg, f, varargin{:});
    end
end
```

The logic is quite simple: If there is no oil/gas data (`krOG`) for a three-phase system available, we must have a water/oil system. If there is no oil/water data (`krOW`) for a three-phase system available, we must have an oil/gas system. Otherwise, all three phases are present and we must set up a full three-phase model. The relative permeabilities for the two-phase water/oil system are set up as follows

```
function [krW, kr0] = relPermW0(sw, f, varargin)
    krW = f.krW(sw, varargin{:});
    if isfield(f, 'kr0')
        kr0 = f.kr0(1-sw, varargin{:});
    else
        kr0 = f.kr0W(1-sw, varargin{:});
    end
end
```

Here, the default is to look for a two-phase table called `kr0` for the oil relative permeability. If this is not present, the user must have supplied an oil/water table from a three-phase model, and we use this instead. The oil/gas system is treated analogously. Finally, for the three-phase model:

```
function [krW, kr0, krG] = relPermWOG(sw, sg, f, varargin)
    swcon = f.swcon;
    swcon = min(swcon, double(sw)-1e-5);

    d      = (sg+sw-swcon);
    ww    = (sw-swcon)./d;
    krW   = f.krW(sw, varargin{:});

    wg    = 1-ww;
    krG   = f.krG(sg, varargin{:});

    so    = 1-sw-sg;
    krow  = f.kr0W(so, varargin{:});
    krog  = f.kr0G(so, varargin{:});
    kr0  = wg.*krog + ww.*krow;
```

Data input to ad-props

To create a fluid object based on ECLIPSE input, we must first be able to read the input file. To this end, MRST has the module `deckformat` which offers functionality for reading, writing, and processing ECLIPSE input files. Previously, we have used the `readGRDECL` and `processGRDECL` functions from the core module to read and process keywords related to grids and petrophysical properties from the GRID section of an input file. The `deckformat` module reads a larger portion of complete input files. The most important function is the reader,

```
deck = readEclipseDeck(fn);
```

which reads the input file `fn` and constructs a structure `deck` with one field for each of the sections in the input file except EDIT and SUMMARY. In passing, we mention that the `deckformat` module offers several functions to construct objects for grids, rock properties, and wells as well as functionality for reading

and processing ECLIPSE *output* files. Hence, MRST can be used to visualize output from ECLIPSE simulations, and it is also possible to use existing ECLIPSE runs to initialize simulation cases in MRST.

To understand the input, we must look in more detail at the input format of the **PROPS** section. These are given as keywords given in capital letters, followed by numbers and terminated by a single /. As an example, let us take an excerpt from the file **BENCH_SPE1.DATA**, which is supplied with MRST and describes input for the SPE 1 benchmark case [204]

```

PROPS =====
----- THE PROPS SECTION DEFINES THE REL. PERMEABILITIES, CAPILLARY
----- PRESSURES, AND THE PVT PROPERTIES OF THE RESERVOIR FLUIDS
-----
-- ROCK COMPRESSIBILITY
-- REF. PRES COMPRESSIBILITY
ROCK      14.7      3.0E-6      /

```

Here, all lines starting with -- are interpreted as comments and the only data contained in this part of the file are the two parameters necessary to describe the compressibility of the rock, i.e., a reference pressure (in units bar) and a compressibility (in units 1/bar). For this particular input file, the relative permeabilities are given in the two keywords **SWOF** and **SGOF**:

```

SWOF
-- SWAT      KRW          KRO          PCOW
0.1200000000      0      1.00000000000000      0
0.1210000000  0.00000011363636      1.00000000000000      0
0.1400000000  0.00000227272727      0.99700000000000      0
0.1700000000  0.00000568181818      0.98000000000000      0
0.2400000000  0.00001363636364      0.70000000000000      0
0.3199999990  0.00002272727261      0.350000004375      0
0.3700000000  0.00002840909091      0.20000000000000      0
0.4200000000  0.00003409090909      0.09000000000000      0
0.5200000000  0.00004545454545      0.02100000000000      0
0.5700000000  0.00005113636364      0.01000000000000      0
0.6200000000  0.00005681818182      0.00100000000000      0
0.7200000000  0.00006818181818      0.00010000000000      0
0.8200000000  0.00007954545455      0.00000000000000      0
1.0000000000  0.00010000000000      0      0
/

```

The four columns represent water saturation (**SWAT**), relative permeability of water (**KRW**), relative permeability of oil (**KRO**), and oil/water capillary pressure (**PCOW**). After processing, the **deck.PROPS** field will have one sub-field containing each of these three data

```

deck.PROPS =
SWOF: {[14x4 double]}
SGOF: {[15x4 double]}
:
ROCK: [1.0135e+05 4.3511e-10 NaN NaN NaN NaN]

```

The careful reader may observe that each of the **SWOF** and **SGOF** fields represent the input data as a cell containing an array. The reason is that the relative permeability input may contain multiple arrays of different length

corresponding to different regions of the reservoir. Each array will then be represented as a cell and the whole function as a cell array.

A fluid object can now be created by calling the `initDeckADIFluid` function from `ad-props`

```
fluid = initDeckADIFluid(deck);
```

The essential part of `initDeckADIFluid` is a generic loop that traverses the fields of `deck.PROPS` and tries to pass each of them onto a constructor function written specifically for each field

```
props = deck.PROPS;
fns = fieldnames(props);
for k = 1:numel(fns)
    fn = fns{k};
    if doAssign(fn)
        asgn = str2func(['assign',fn]);
        try
            fluid = asgn(fluid, props.(fn), reg);
        catch %#ok
            warning(msgid('Assign:Failed'), ...
                    'Could not assign property ''%s''.', fn)
        end
    end
end
fluid = assignRelPerm(fluid);
```

The second line extracts a list with the names of all fields present, the fifth line checks the field name against a list of field names that either have not been implemented yet or should for some reason not be processes. The sixth line makes a function pointer by appending the field name to the string “assign”, in our case creating function pointers to `assignROCK`, `assignSWOF`, and `assignSGOF`. These functions are then called in the eighth line, with the fluid object `fluid` and the appropriate fluid data as input. (The third input argument, `reg`, represents region information). Let us look at the implementation of rock compressibility. If we for simplicity disregard region information, the model for rock compressibility is set up as follows

```
function f = assignROCK(f, rock, reg)
    [cR, pRef] = deal(rock(2), rock(1));
    f.cR = cR;
    f.pvMultR = @(p)(1 + cR.* (p - pRef));
end
```

Although rock compressibility, strictly speaking, is not a *fluid* relationship, it has for historic reasons been bundled with the saturation and pressure-dependent fluid and rock-fluid relationships since it represents a purely pressure-dependent quantity and will enter the flow equations in the same way as the other elements of the fluid object. Notice also that with the generic

setup in the `initDeckADIFluid`, all you have to do to support a new keyword `MYKEY`, is to implement the function `assignMYKEY` and place it in the right directory of the `ad-props` module.

Assigning relative permeabilities is somewhat more involved. If we disregard code lines having to do with the region information (i.e., that the model can use different relative permeability functions in different sub-regions of the grid), the constructor takes the following form

```
function f = assignSWOF(f, swof, reg)
    f.krW = @(sw, varargin) krW(sw, swof, reg, varargin{:});
    f.krOW = @(so, varargin) krOW(so, swof, reg, varargin{:});
    f.pcOW = @(sw, varargin) pcOW(sw, swof, reg, varargin{:});
    swcon = cellfun(@(x)x(1,1), swof);
    f.sWcon = swcon(1);
end
```

That is, we add three function pointers to the fluid object implementing the relative permeability of water, the relative permeability from an oil/water displacement, and the capillary pressure of between oil and water. Let us take the `krOW` function as an example,

```
function v = krOW(so, swof, reg, varargin)
    satinx = getRegMap(so, reg.SATNUM, reg.SATINX, varargin{:});
    T = cellfun(@(x) x(:,[1,3]), swof, 'UniformOutput', false);
    T = extendTab(T);
    v = interpReg(T, 1 - so, satinx);
end
```

The first line sets up a map that lets us select the correct saturation functions if there are multiple regions. The next line extracts the first and the third column of the input data, i.e., the `SWAT` and `KRO` columns. Here, we use `cellfun` in combination with an anonymous function to extract individual elements, in the same way as we used `arrayfun` to extract simulation results from the `sol` array of structures in the previous section. The saturation values in the relative permeability tables may not necessarily span the whole unit interval, and in the third line we add a constant extrapolation to ensure that we can use `T` to extrapolate for all legal saturation values. The last line calls an interpolation function that uses region information to pick the correct relative permeability function from the cell array and then calls an appropriate interpolation function. If `T` had been a simple array, the last line would have amounted to

```
F = griddedInterpolant(T(:,1), T(:,2), 'linear', 'linear');
v = F(1-so);
```

The `krW` and `pcOW` functions are implemented exactly the same way, with the obvious modifications, and so are the functions in `assignSGOF`.

Example: SPE 1, SPE 3, and SPE 9

In 1981, Aziz S. Odeh [204] launched an initiative for independent comparison of the state-of-the-art for 3D reservoir simulation of black-oil models. To represent a good cross-section of solution methods used throughout industry, seven companies were selected, each participating with their own simulator. This type of comparison was well received by both industry and the academic community. Over the next two decades, the Society of Petroleum Engineering published a series of nine additional studies, which are today known as the ten SPE comparative solution projects. In each study, a number of oil companies, software providers, research institutes, universities, and consultants participated on a voluntarily basis, reporting their best attempt to solve a specified problem. Altogether, these ten problems constitute a rich source of test problems that can be used to verify and validate various multiphase models and solution strategies. You have already seen data from the tenth and last project used multiple times throughout this book.

In this example, we use input decks for the SPE 1 [204], SPE 3 [130] and SPE 9 [131] comparison projects to demonstrate how MRST can be used to set up and compare different fluid models. If you have not already done so, the necessary files can be downloaded by use of MRST's data set manager:

```
mrstDatasetGUI;
```

This will place the data sets on a standard path, so that they can be read and processed using the following set of commands (with the obvious modifications for SPE 3 and SPE 9):

```
mrstModule add deckformat ad-core ad-props
pth = getDatasetPath('spe1');
fn = fullfile(pth, 'BENCH_SPE1.DATA');
deck = readEclipseDeck(fn);
deck = convertDeckUnits(deck);
f = initDeckADIFluid(deck);
```

This produces a fluid object with a number of fields (here for SPE 1):

```
krW: @(sw,varargin)krW(sw,swof,reg,varargin{:})
krOW: @(so,varargin)krOW(so,swof,reg,varargin{:})
pcOW: @(sw,varargin)pcOW(sw,swof,reg,varargin{:})
sWcon: 0.1200
krG: @(sg,varargin)ireg(Tkrg,sg,varargin{:})
krOG: @(so,varargin)ireg(Tkro,sgas(so),varargin{:})
pcOG: @(sg,varargin)ireg(Tpcog,sg,varargin{:})
:
relPerm: @(sw,sg,varargin)relPermWOG(sw,sg,f,varargin{:})
```

Here, we recognize relative permeability of water, oil relative permeability of an oil/water system, oil/water capillary pressure, connate water injection, etc. The last function is a generic interface for evaluating relative permeabilities

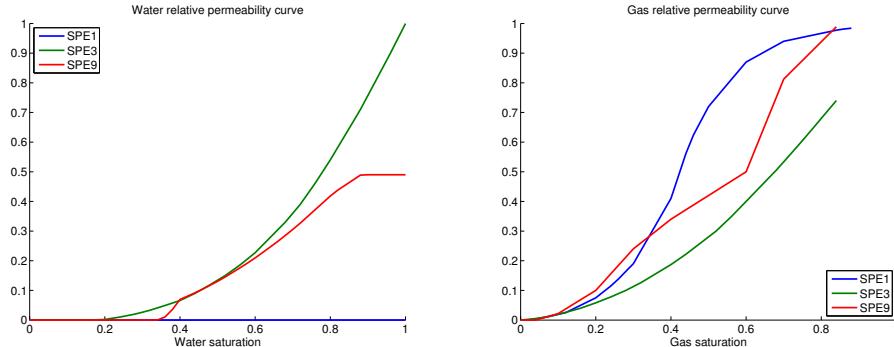


Fig. 11.6. Relative permeabilities for water and gas for the fluid models from three of the SPE Comparative Solution Projects.

of all phases in the three-phase system. Let us look at the two-phase relative permeabilities krW and krG . These can be evaluated by a call on the form

```
s = linspace(0,1,51)';
krW = f.krW(s);
```

The resulting curves are plotted in Figure 11.6 (extrapolated krG values are not plotted for saturation for $S_w < S_{wc}$). SPE 1 is represented as a three-phase system in this particular input file, but it is really a two-phase gas/oil problem describing gas injection into an undersaturated oil, i.e., an oil that can dissolve more gas (we will discuss this sort of PVT behavior in the next section). The water relative permeability is therefore (virtually) zero over the whole saturation interval. For the SPE 3 model, both relative permeability curves have monotone derivatives similar to what we have seen in the analytic models discussed in Section 8.1.4. This is not the case for the gas relative permeability of SPE 1, which has an S-shape similar to the fractional flux functions we have seen in two-phase systems. For the SPE 9 model, the relative permeability curves have significant kinks, which will contribute to increase the nonlinearity of the discretized flow equations.

Figure 11.7 shows the corresponding relative permeability curves for two-phase oil/water and gas/oil systems and the resulting interpolated three-phase oil relative permeability curve. The plots were generated as follows:

```
[sw, sg] = meshgrid(linspace(0,1,201)); so=1-sw-sg;
[~, kr0, ~] = f.relPerm(sw(:,sg(:)));
kr0 = reshape(kr0,size(sw)); kr0(sw+sg>1)=nan;

[mapx,mapy] = ternaryAxis('names',{'S_w';'S_g';'S_o'});
contourf(mapx(sw,sg,so), mapy(sw,sg,so), kr0, 20)
```

For SPE 1 and SPE 3, the $krOW$ and $krOG$ curves representing two-phase displacements coincide and hence the interpolated krO surface is smooth and

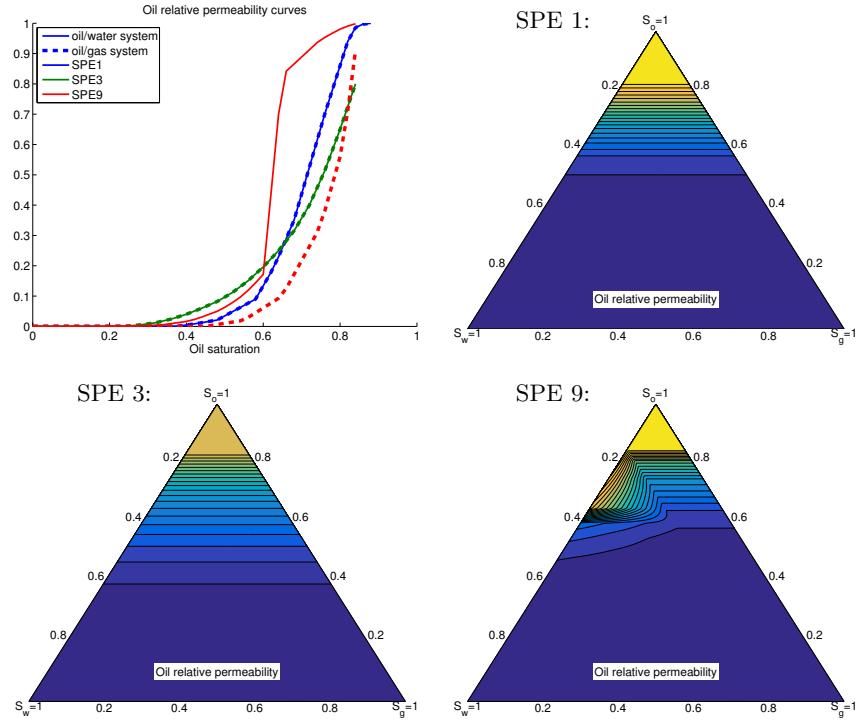


Fig. 11.7. Three-phase relative permeabilities of oil for three of the SPE Comparative Solution Projects generated by linear interpolation of two-phase curves from oil/water and oil/gas displacements (upper left plot).

monotone along straight lines in phase space. This is not the case for SPE 9. Not only are the two-phase curves significantly different, but the k_{rOW} curve has a very steep gradient. As a result, we observe kinks and it is also possible to identify straight lines in the phase diagram along which k_{rO} is non-monotone, see e.g.,

```
sw = linspace(0,.6,21); sg = linspace(0,.4,21);
[~,kr0] = f.relPerm(sw',sg');
```

As is warned about in [226], abrupt changes in relative permeability values of small saturation intervals can be the source of significant convergence problems when solving the discretized flow equations and may cause the simulator to slow down. Like ECLIPSE, MRST honors the data input and does not attempt to overcome convergence problems by smoothing kinks or retabulating the input data.

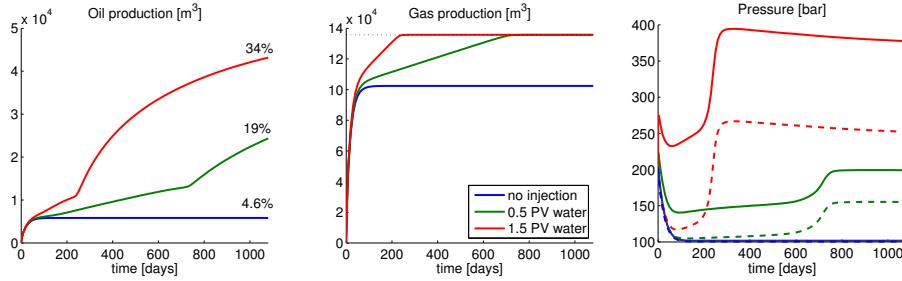


Fig. 11.8. Conceptual three-phase simulation with no mass transfer between the phases. The reservoir is initially filled with gas on top, oil in the middle, and water at the bottom. Fluids are produced from the middle. The plots show results of simulations with no water injection and with water injection into the bottom layer giving a total injected amount of 0.5 and 1.5 pore volumes, respectively. The plots to the right show average pressure in the injector cell (solid lines) and field average pressure (dashed lines).

Example: A simple three-phase simulator

Having established how to model three-phase relative permeabilities, it is a simple exercise to extend the two-phase simulator to three-phases under the assumption of no mass transfer between the three phases in the reservoir. This assumption is not very realistic, but is introduced to isolate and illustrate certain aspects of multiphase flow. To exemplify, we consider a setup similar to the one discussed in Section 11.1 with the upper 1/3 initially filled with gas ($S_g = 1 - S_{wc}$), the middle 1/3 filled with oil ($S_o = 1 - S_{wc}$), and the bottom 1/3 is filled with water. Fluids are produced from a cell at northeast corner of the middle section by setting the pressure in this cell to be 100 bar, which is approximately 100 bar lower than the initial average reservoir pressure. We change the fluid model so that water is twice as viscous as oil, and assume that relative permeabilities follow the SPE 3 fluid model.

When the producer starts depleting the reservoir pressure, the compressed gas will expand so that a large fraction of it is driven out of the reservoir through the oil layer below. Gas is much more mobile than oil and will flow through the oil rather than displacing it, and hence only a minor fraction of the oil will be recovered. To increase oil recovery, we therefore inject water into the water zone at the lower southwest corner. Code to simulate this case is given in `threePhaseAD.m` in the `book` module. To run a comparison of no water injection and injection of 0.5 and 1.5 pore volumes of water

```
pvi=0; threePhaseAD; pvi=.5; threePhaseAD; pvi=1.5; threePhaseAD
```

Figure 11.8 reports observed hydrocarbon production and pressure behavior. Without water injection, the reservoir pressure is quickly depleted and reaches a steady state after approximately 100 days, giving a total gas production

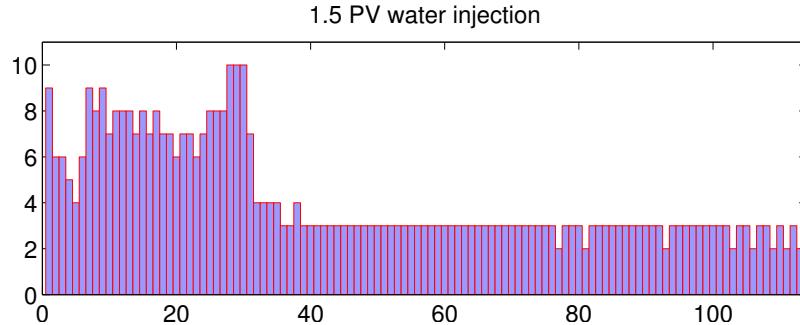


Fig. 11.9. Number of nonlinear iterations for the conceptual three-phase simulation.

of 75% and an oil production of 4.6% of the volumes initially in place. By injecting 0.5 pore volumes of water, we are able to maintain reservoir pressure and hence higher oil production in the period until all the gas is depleted and the injected water breaks through in the producer after approximately 740 days. Towards the end of this period, the pressure increases significantly as the water front approaches the producer and then stabilizes at a plateau. When gas disappears and oil is the most mobile phase, the oil production increases abruptly, but then decays slowly as a result of gradually increasing water cut in the producer. Similar behavior can be observed when injecting 1.5 pore volumes of water. However, since the injection rates are three times higher, water breakthrough occurs much earlier and we also observe pressures much higher than the initial field pressure.

If we try to repeat the simulation with a total injection of two pore volumes, the simulator fails to converge within fifteen iterations in the 23rd time step, right before the water front breaks through in the producer. A similar increase in the number of iterations towards water breakthrough can also be observed for the other water injection cases, see Figure 11.9. It is well known that Newton's method may experience convergence problems if saturations (or compositions) change too much during a single time step. We have already discussed this issue in Section 10.2.2, where we also introduced a reactive strategy for chopping of the time step.

COMPUTER EXERCISES:

79. Extend the simulator in `threePhaseAD.m` to include the reactive time-step control discussed in Section 10.2.2 and try to extend it so that it also uses changes in saturation as a criterion to chop time steps. Does this enable you to run a simulation with more than two pore volumes injected?

11.3 PVT behavior of petroleum fluids

In our discussion so far, we have assumed that there is no mass transfer between phases and that each fluid phase has constant compressibility, but otherwise constant fluid properties. This is a gross simplification. Pressure and temperature will not only affect the compressibility and viscosity of each fluid phase, but also determine which phases are present and how the different chemical components in the reservoir fluids distribute among the aqueous, oleic, and gaseous phases at reservoir and surface conditions. Hydrocarbon fluids can come in a wide variety of forms, including the bitumen (semi-solid rock form of crude oil) of the Athabasca oil province in Canada and heavy (dead) oils containing almost no light hydrocarbon components, which neither will be discussed in any detail. Instead, we focus on hydrocarbon fluids that have a significant gas component at surface conditions. That is, medium oils containing lighter components that are liberated as free gas at surface conditions; volatile oils that shrink significantly when liberating free gas; rich gases that condense liquid at reduced pressure; wet gases that condensate fluids at lower temperatures; and dry gases that mainly consist of light hydrocarbons and stay in single-phase vapor form.

11.3.1 Phase diagrams

We have previously discussed equations-of-state modeling the relationship among pressure, volume (density), and temperature. For reservoir fluids it is more common to use *phase diagrams*, which in this setting are a type of chart showing pressure and temperature conditions at which thermodynamically distinct phases occur and coexist at equilibrium. In this section we will discuss such diagrams for reservoir fluids.

Single-component substances

As a starting point, however, let us look at a typical phase diagram for a substance consisting of a single component. To explain this diagram, we must first look at Gibbs' phase rule for a system without chemical reactions, which states that the number of independent degrees of freedom F that must be specified to determine the intensive state of a system is given by

$$F = 2 + n_c - n_p,$$

where n_c is the number of components, and n_p is the number of phases. For a single-component system ($n_c = 1$), we have $F = 3 - n_p$, and hence there can at most be three phases, which corresponds to zero degrees of freedom and hence must coexist in a single point called the *triple point*, at which pressure and temperature are uniquely determined. If two phases coexist, there is one degree of freedom, which means that pressure can be chosen freely if temperature is

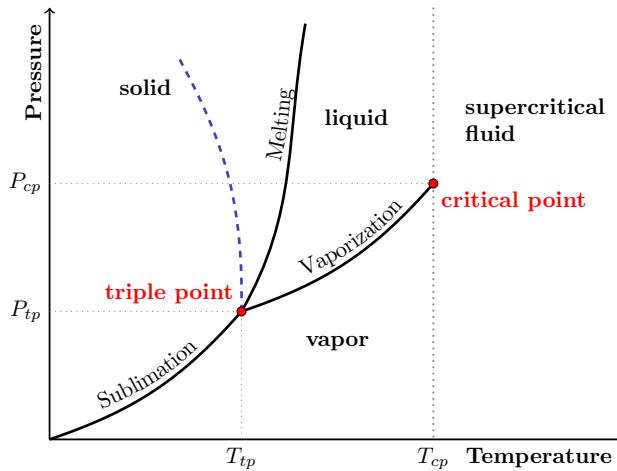


Fig. 11.10. Conceptual phase diagram for a single-component substance. The dashed blue line illustrates the melting curve for water, which unlike most other fluids has negative slope.

fixed, and vice versa. When the substance is in a single phase state, both pressure and temperature can be chosen independently.

Figure 11.10 illustrates the resulting phase diagram. In the diagram, the three single-phase regions corresponding to solid, liquid, or vapor phase are separated by the sublimation, melting, and vaporization curves. Along these curves, two phases coexist in equilibrium. Across the curves, various properties of the substance (compressibility, viscosity, etc) will change abruptly and have discontinuous derivatives with respect to temperature and pressure. At the triple point, all phases exist in equilibrium. For temperatures below the critical point T_{tp} , the substance will be in vapor phase if the pressure is below the sublimation curve and in solid state for pressures above. For pressures and temperatures along the sublimation curve, vapor and solid are in equilibrium. For temperatures between the triple point and the critical point ($T_{tp} < T < T_{cp}$), the substance will be in vapor state if the pressure is below the vaporization curve, in liquid state if the pressure is above the vaporization curve, and in solid state if the pressure is above the melting curve.

The liquid–vapor curve terminates at the critical point. The critical temperature T_{cp} is the temperature at and above which vapor of the substance cannot be liquefied, no matter how much pressure is applied. Critical temperatures depend on the size of the molecule and will hence vary a lot for hydrocarbons, from low temperatures for small molecules to high temperatures for large molecules. For water, the critical temperature 374° C , whereas carbon dioxide has a much lower critical temperature of 31.2° C . Likewise, the critical pressure P_{cp} is the pressure required to liquefy a gas at its critical temperature, or in other words, the pressure above which distinct liquid and

gaseous phases do not exist. The critical pressure of water is 217.7 atm and for CO₂ it is 73.0 atm. For temperatures above T_{cp} , the substance is said to be in *super-critical state*, i.e., in a state in which the liquid and gaseous phases are indistinguishable. This is the state in which CO₂ is usually found when stored in a deep geological formation (approximately below 800 meter).

For most substances, the melting curve separating liquid and solid phase has a positive derivative with respect to temperature; the denser the molecules of the substance are brought together, the higher temperature is required to break them apart to form a liquid state. Water is one exception to this rule.

For hydrocarbon recovery, however, we will primarily be interested in the vaporization curve. To better understand the phase behavior for multicomponent systems, we will briefly explain an experimental setup for determining this curve using a high-pressure cell as illustrated in Figure 11.11. Initially, the cell is filled with liquid hydrocarbon at the top and mercury at the bottom. As mercury is gradually removed, the liquid hydrocarbon expands and the pressure drops. This continues until the first bubble of gas comes out of the solution as we reach the bubblepoint (point 2 in the figure). Beyond this point, the gas phase expands the volume rapidly while keeping the pressure constant (point 3 in the figure). Eventually, we reach the dewpoint, where the last liquid droplet vaporizes (point 4 in the figure). As more mercury is removed, the gas expands while the pressure decreases. If we repeat the experiments for multiple temperatures, we get the diagram shown in the right plot. By projecting the locus of bubblepoints, which we henceforth will refer to as the bubblepoint curve, and the locus of dewpoints, which is called the dewpoint curve, we obtain the vaporization curve in the pressure-temperature diagram.

Binary substances

Because hydrocarbon reservoir fluids generally are multicomponent systems made up of a wide range of chemical substances, their phase diagrams will be very complicated. Gibbs' phase rule will obviously give us little guidance since the number of actual phases occurring in the reservoir is much less than the maximum phases that may exist. In the black-oil model, the hydrocarbon fluids are considered as a two-component system. (Notice that this distinction between an oil and a gas component is only relevant when the reservoir is in a state where both can coexist.) For binary substances, the number of degrees of freedom for a two-phase system is $F = 2$ and hence two phases could in principle exist for arbitrary combinations of temperature and pressure although this is not the case in practice. To describe the reservoir fluids, we utilize a bubblepoint/dewpoint projection similar to the one we studied for the single-component fluid above. This projection is defined for a constant mole fraction of one of the components, and hence the phase diagram will depend on the composition of the binary mixture. In the projected plot, the bubblepoint and dewpoint curves will no longer coincide along a single vaporization curve. Once the bubblepoint is reached, the expansion is accompanied

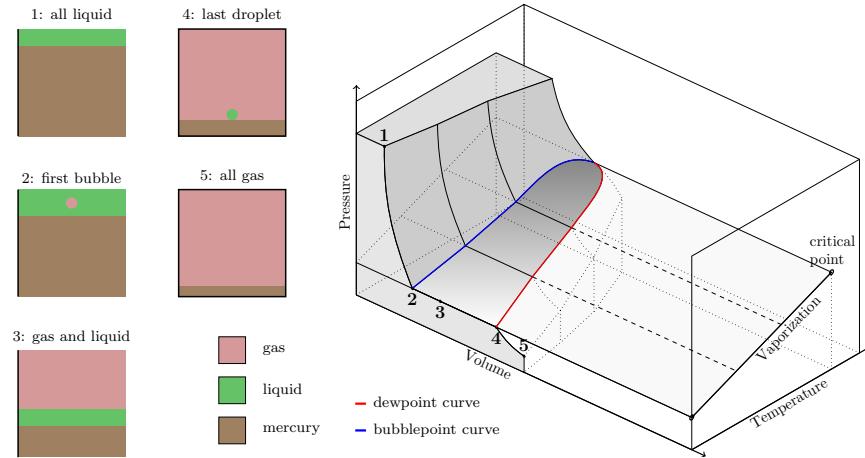


Fig. 11.11. PVT behavior of a single-component hydrocarbon substance. The left figures illustrate measurement of pressure-volume behavior in a high-pressure cell at fixed temperature. (The mercury is connected to a pump that can inject or extract this fluid to control pressure. Mercury has high surface tension and will hence be non-wetting for most container surfaces like glass.) The right plot shows the measurements plotted in a PVT diagram, with the bubblepoint and vaporization curves projected onto a pressure-temperature diagram.

by a pressure decrease since the gaseous phase has a different chemical composition than the oleic phase. The projection hence produces a temperature and pressure envelope inside which the binary mixture forms two phases as shown in Figure 11.12.

At pressures and temperatures above the bubblepoint curve, the binary mixture will form a liquid phase. Likewise, a gaseous phase is formed for all points located above or to the right of the dewpoint curve. The bubblepoint and dewpoint curves meet in the *critical point*, at which the phase compositions and all phase properties are identical. Notice that the critical point does not necessarily coincide with the maximum temperature the two-phase envelope. The maximum temperature is called the *cricondentherm* and the maximum pressure is called the *cricondenbar*. The critical point represents a mathematical discontinuity in the system, and the phase behavior is generally difficult to define near this point, meaning that it is not obvious whether a (slight) pressure decline will cause the formation of bubbles of vapor or droplets of liquid.

11.3.2 Reservoir types and their phase behavior during recovery

In petroleum literature, the phase diagrams discussed above are sometimes used to classify reservoirs according to the type of phase behavior that takes place during (primary) production. To motivate this classification, and the

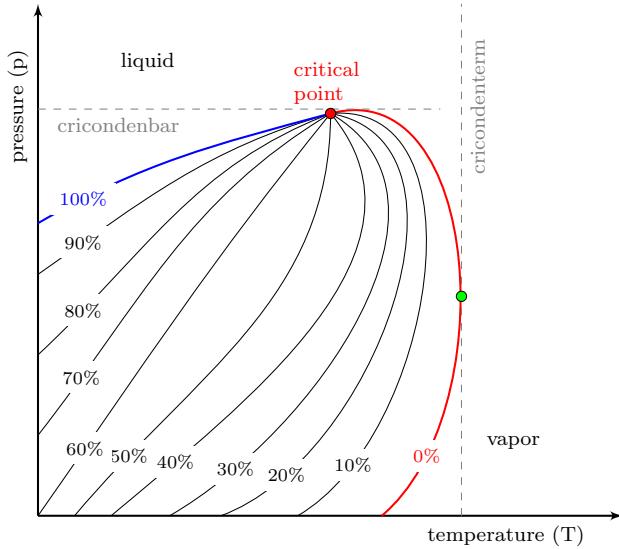


Fig. 11.12. Pressure-temperature diagram for a binary system of fixed composition. The bubblepoint curve is shown in blue and the dewpoint curve shown in red. The lines inside the two-phase envelope represent constant percentages of liquid. The diagram is specific to a given chemical composition of the binary mixture and will change if this composition changes. e.g., if the components are extracted at different rates.

choice of physical variables and closure relationships to be discussed in the next section, we start by a schematic of a typical production setup for a depleting reservoir as shown in Figure 11.13.

When a well is opened to start production of hydrocarbons, the pressure inside the reservoir will start depleting. As long as no other fluids are injected into the reservoir, the temperature of the reservoir fluids will remain (almost) constant at the value given by the geothermal gradient of the area because of the large thermal inertia of the reservoir. Inside the reservoir, the extraction of hydrocarbons will therefore take place along a vertical line in the pressure/temperature diagram. Also when fluids are injected to displace the resident hydrocarbons, temperature effects will in many cases be so small that the temperature can be considered as constant (near the producing well) in the reservoir model. This means that various types of reservoirs can be classified by the location of the initial reservoir pressure and temperature with respect to the bubblepoint/dewpoint envelope. If the temperature of the reservoir is less than the critical temperature ($T_r < T_c$) of the hydrocarbon mixture, the reservoir is said to be an oil reservoir. Conversely, if the temperature is larger than the critical temperature ($T_r > T_c$), the reservoir is said to be a gas reservoir. In both cases, the reservoir may contain hydrocarbons in both liquid and vapor form, depending on the type of the reservoir.

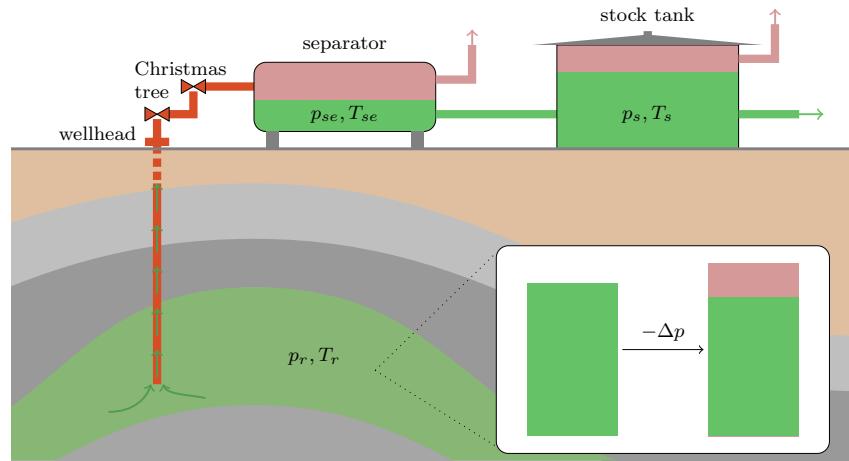


Fig. 11.13. Illustration of pressure and temperature changes resulting in liberation and expansion of gas and shrinkage of oil during production of a hydrocarbon reservoir containing both oil and gas. Stock-tank conditions vary but will roughly be 1 atm and 20° C.

As the hydrocarbons leave the reservoir and flow to the surface, they will undergo a series of changes that reduce the pressure and temperature to surface conditions. These changes will generally cause liberation of gas, expansion of the gaseous phase, and shrinkage of the liquid oleic phase. Pressure reductions come in two types: In *flash liberation*, gas comes out of solution as a result of a sudden pressure drop, and then remains confined with the remaining oil. In *differential liberation*, gas is liberated from the oil as pressure is gradually decreased and is continuously removed from the oil. How the reservoir fluids contained in the well stream are brought from reservoir pressure and temperature to surface condition can have a large effect on how much oil and gas that is recovered from a given quantity of reservoir fluids. Traditionally, oil has had much higher economic value than gas, which often has been considered as a waste product. It has therefore been highly important to design a production system that maximizes the stock-tank oil yield.

There are many mechanisms inducing pressure reduction: friction and gravity forces acting in the opposite direction of the flow, loss of kinetic energy due to expansion, contraction in the fluid area, flow through constrictions, etc. At the surface, the fluids will pass through the Christmas tree mounted on top of the well head, whose main purpose is to control the flow of reservoir fluids out of the well. The Christmas tree has isolation valves to stop the fluid flow for maintenance or safety purposes, flow lines, and chokes that control the flow into the downstream production system. Inside the production system, the fluids can continue into one or more pressure vessels used to separate the produced fluids into oil, gas, and water components, remove . The oil compo-

ment may continue to a stock tank for temporary storage, inside which further gas liberation may take place.

We already know that the pressure drop inside the reservoir and the pressure drop from the reservoir to the bottom hole both depend on flow rate; this follows from Darcy's law and the Peaceman well model, respectively. Similar relationships between pressure drop and flow rate can be developed for the other parts of the flow system from the bottom hole to the stock tank; that is, pressure loss in the tubing, in the Christmas tree and flow lines, across the choke, and in the separator units. These relationships are used by production engineers to optimize the hydrocarbon recovery. For a more detailed discussion, we refer the reader to [116].

Gas reservoirs

All reservoirs in which the initial temperature is higher than the critical temperature are called gas reservoirs. Methane is by far the dominant hydrocarbon component in gas reservoirs. If the initial temperature is higher than the cricondentherm, the hydrocarbon mixture will be in a single-phase gaseous state and will continue to be so during depletion. As the reservoir is depleted, the composition of the produced fluids will not change, but the hydrocarbon mixture may enter the two-phase envelope on its way to the surface, depending upon the cooling process, as illustrated in the left plot of Figure 11.14. If a single-phase reservoir produces liquid hydrocarbons at the surface (condensate), the reservoir is said to be a *condensate gas, wet gas, or rich gas* reservoir. Conversely, if the temperature-pressure path followed during production does not penetrate the two-phase envelope, all hydrocarbons produced at the surface will be in gas phase and the reservoir is said to be a *dry gas* reservoir. Dry gas reservoirs seldom contain hydrocarbons with carbon number five or higher and typically consists of C₁, C₂, N₂, CO₂, and H₂S. No hydrocarbon separation is needed at the surface since the well stream consists of gas only, but separators/dehydrators may be needed to remove produced/condensed water. Although wet gas reservoirs mainly consist of methane, they also contain both intermediate hydrocarbons (ethane to hexane) as well as a significant fraction of hydrocarbons with carbon number seven or higher. The condensation of liquid hydrocarbons at the surface is a result of cooling, but whether the well stream is brought to surface conditions in one big flash or through a series of smaller flashes does not have a large effect on the relative amounts of gas and liquid observed at the surface.

Unlike dry/wet gas reservoirs, *dewpoint* or *retrograde gas condensate* reservoir contain a significant fraction of heavier hydrocarbon components (C₇₊). The right plot in Figure 11.14 shows such a hydrocarbon mixture, which is initially in (an indeterminate) vapor state and will continue to be so as the reservoir is depleted towards point A on the dewpoint curve. During this initial decline, the composition of the produced fluid remains constant. Below point A, a liquid oleic phase containing heavier hydrocarbons will condense

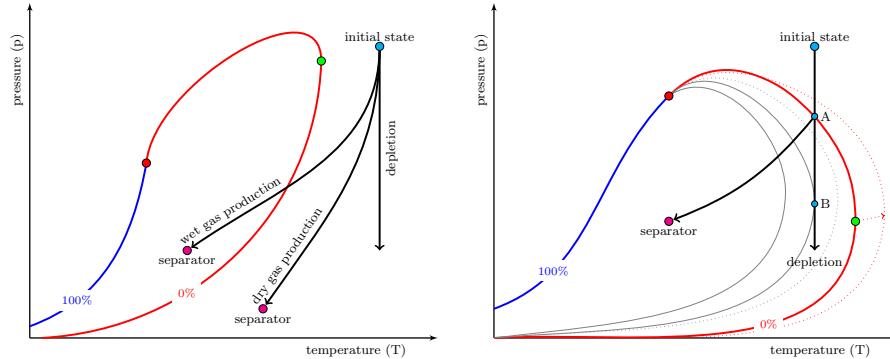


Fig. 11.14. Phase diagrams for gas reservoirs. In a single-phase dry/wet gas reservoir (left), the temperature is above the cricondentherm. In a retrograde gas condensate reservoir (right), the temperature is above the critical temperature, but below the cricondentherm. During the retrograde depletion below the dew-point curve, the composition of the mixture will change, and this induces a change in the diagram, as indicated by the dotted lines.

out of the vapor as a dew or fog and continue to do so until the point *B* of maximum liquid percentage. This process is called retrograde, since one normally would see vaporization and not condensation when the pressure decays. The condensed liquid is immobile at low saturations and generally less mobile than the vapor phase. Hence, more light hydrocarbons will be recovered from the reservoir, giving an increased gas-oil ratio at the surface. This changes the composition of the hydrocarbon mixture, increasing the fraction of heavier hydrocarbons, which in turn induces a shift in the two-phase envelope toward the right, as indicated by the dotted lines in the figure. This aggravates the loss of heavier hydrocarbons, which are retained as immobile droplets inside the rock pores. However, if production is continued, the system will sooner or later reach a point of maximum liquid percentage, below which the condensed liquid will start to vaporize. This improves the mobility of the liquid components and leads to a decrease in the surface gas-oil ratio. For a dewpoint reservoir, the exact design of how the reservoir fluids are brought to surface conditions can have a large impact on the amount of stock-tank oil and gas. This may call for a complex network of separators in which separated vapor or liquid is reinjected into the liquid-vapor mixture to better distribute heavier hydrocarbon components to the stock-tank oil and lighter components to the final gas stream.

In closing, we remark that the distinction between condensate and oil is somewhat artificial and has more to do with how the liquid hydrocarbons have been recovered than their chemical composition. Liquid hydrocarbons in the stock tank are called oil if they originate from a liquid oleic state in the reservoir and condensate when they have been condensed from a gaseous state.

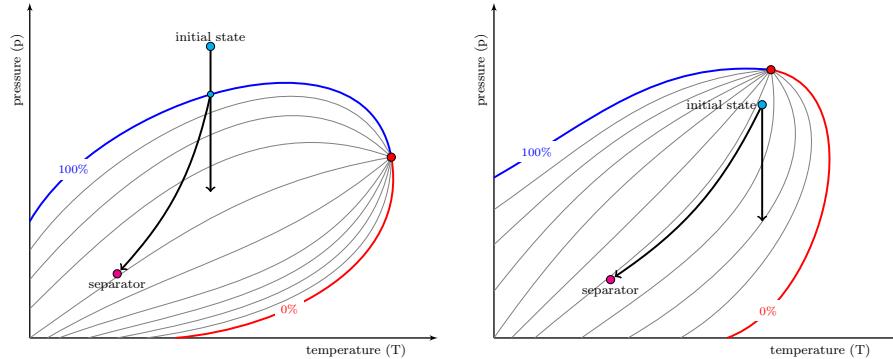


Fig. 11.15. Phase diagram for bubblepoint oil reservoirs. The left plot shows a black-oil reservoir with undersaturated initial oil and the right plot a volatile oil reservoir.

Bubblepoint oil reservoirs

A reservoir having initial temperature below the cricondentherm is characterized as an oil reservoir. Unlike the dead oil considered in Section 11.1, which either contained a constant amount of dissolved gas or had lost all its components, the oleic phase in bubblepoint reservoirs consists of a *live oil* containing dissolved gas, which may be released at lower pressures inside the reservoir and at the surface. This type of reservoir is also called a *solution-gas* reservoir, and will be the topic for most of the simulation examples discussed later in the chapter. Although somewhat artificial and imprecise, it is common to subdivide oil reservoirs into two categories: *Black-oil* reservoirs are mainly made up of a variety of hydrocarbons, including heavy and non-volatile components (C_{7+}). *Volatile* reservoirs contain fewer heavy components and have more intermediate hydrocarbon molecules (ethane to hexane) and therefore tend to shrink dramatically when gas is liberated. Black-oil reservoirs tend to have much less shrinkage. The main property that distinguishes the two categories, however, is the presence of vaporized oil in the gaseous phase, which is assumed to be negligible in black-oil reservoirs and significant in volatile reservoirs. This might be a bit confusing since we later will use *black-oil* equations to model both types of reservoirs. The reason is most likely historic. Traditionally, the largest and most profitable reservoirs in the world have predominantly been of black-oil type, and the black-oil equations were developed to model these. Volatile reservoirs are typically found at greater depths, and as exploration goes deeper, more reservoirs of this type have been discovered and entered production. Volatile reservoirs contain less oil per volume, but since the reservoir fluids are more mobile because of lower oil viscosity and higher dissolved-gas content, this type of reservoir can still be more economically attractive than a black-oil reservoir.

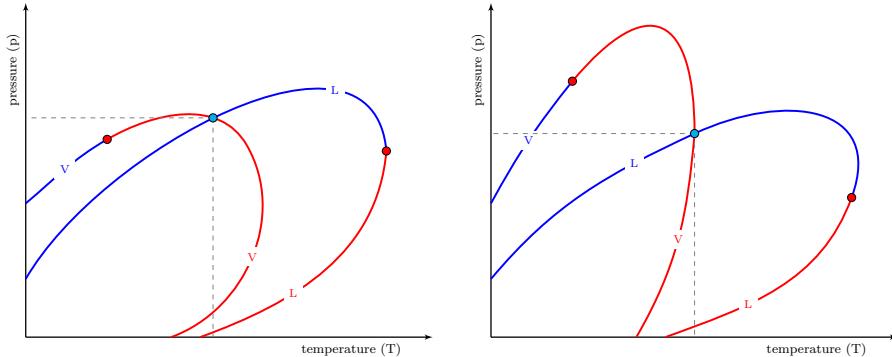


Fig. 11.16. Phase diagrams for a reservoir at saturated conditions so that liquid oil is in equilibrium with a gas cap containing vaporized oil. In the diagrams to the left, the gaseous phase will exhibit retrograde behavior and in the diagrams to the right, it will not.

If the pressure is above the bubblepoint curve as shown in the left plot of Figure 11.15, the hydrocarbon mixture of the oil reservoir is in a liquid state usually referred to as *undersaturated* since all the available gas has been dissolved in the oleic phase. However, when the first liquid is brought to the surface, it liberates free gas. As production continues, the pressure will decline toward the bubblepoint curve. Below the bubblepoint, gas will be released from the oleic phase as vapor bubbles that gradually form a free gaseous phase. When the gaseous phase exceeds its residual saturation, it will start to flow toward the producer. At the surface, the gas will consist of both expanded free gas and gas liberated from the oil during production. Oftentimes, surface facilities will have to limit the gas rate, which in turn can reduce the oil rate below an economic level unless secondary mechanisms like water inject are applied to maintain the reservoir pressure and push the oleic phase toward the producer (as we saw for the dead oil system in Section 11.2).

The volatile reservoir shown to the right in Figure 11.15 is initially in an undersaturated state inside the two-phase envelope. The reservoir fluids thus consist of a liquid oleic phase which exists in equilibrium with a gaseous phase containing gas condensate. Because the gaseous phase has a different composition and is lighter than the oleic phase, the gaseous phase will form a gas zone or gas cap overlying the liquid oil zone and will have a phase diagram that is largely different from that of the oleic phase. Equilibrium between the two phases implies that the oleic phase must be at its bubblepoint, whereas the gaseous phase must be at its dewpoint and may either exhibit retrograde or non-retrograde behavior, as illustrated in Figure 11.16. Notice also that volatile reservoirs are not very different from retrograde gas condensate reservoirs for temperatures close to the critical temperature.

We end our discussion about reservoir classification by presenting two comparisons of the different reservoir types. Figure 11.17 shows a schematic with

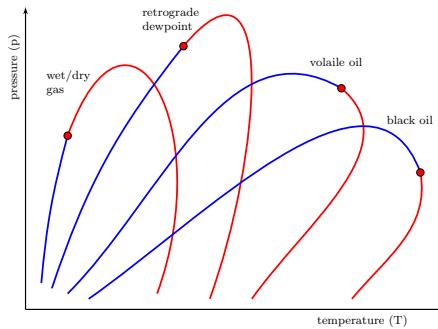


Fig. 11.17. Sketch of phase diagrams for different types of hydrocarbon reservoirs.

Table 11.1. Petroleum fluids and their characteristics at initial conditions or bubble-point pressure (adapted from petrowiki.org/Oil_fluid_characteristics).

Characteristic	Oils			Gases	
	Heavy oils and tars	Black oils	Volatile oils	Gas condensates	Wet and dry gases
Mol. weight	210+	70 to 210	40 to 70	23 to 40	<23
Oil color in stock tank	black	brown to light green	greenish to orange	orange to clear	clear
Stock-tank °API	5 to 15	15 to 45	42 to 55	45 to 60	45+
C ₇₊ fraction (mol%)	>50	35 to 50	10 to 30	1 to 6	0 to 1
Reservoir temperature °F	90 to 200	100 to 200	150 to 300	150 to 300	150 to 300
Saturation pressure [psia]	0 to 500	300 to 5 000	3 000 to 7 500	1 500 to 9 000	—
Initial B _o [RB/STB]	1.0 to 1.1	1.1 to 1.5	1.5 to 3.0	3.0 to 20.0	20+
Initial R _s [scf/STB]	0 to 200	200 to 900	900 to 3 500	3 500 to 30 000	30 000+
Initial R _v [STB/MMscf [†]]	0	0 to 10	10 to 200	50 to 300	0 to 50
Max vol% liquid [‡]	100	100	100	0 to 45	0

† at bubble point pressure,

‡ at constant composition expansion of reservoir fluid

the different phase diagrams plotted relative to each other. This sketch is by no means accurate, but illustrates how the phase diagrams move to the right when the hydrocarbon mixture contains (a larger fraction of) heavier hydrocarbon components. Table 11.1 reports a number of characteristic properties for different reservoir types.

11.3.3 PVT and fluid properties in black-oil models

The phase diagrams discussed in the previous section are useful to classify reservoir fluids and visualize the pressure-temperature path from reservoir to surface conditions and aid the reservoir engineer when deciding the depletion process, designing top-side processing facilities (separators, etc), and devising appropriate strategies for improved oil recovery (e.g., water injection or reinjection of gas to maintain pressure, etc). The phase diagrams are also useful when setting up the fluid sampling program used to determine fluid parameters describing PVT behavior in flow equations.

As we have seen already in Section 8.2.3 on page 250, the fluid and PVT description in a black-oil model tries to mimic the behavior illustrated in Figure 11.13 and discussed in detail in the previous section. That is, the composition and density of the oil and gas components are defined at surface conditions (where they easily be measured), and then the following three parameters are used to determine the density of the oleic and gaseous phases at reservoir conditions:

B_o – The oil formation-volume factor is defined as the volume of oil in liquid phase that must be recovered from the reservoir to produce one volume unit of stock-tank oil; in other word, $B_o = V_o/V_{os}$. A reservoir volume V with porosity ϕ and oil saturation S_o thus contains $\phi VS_o/B_o$ stock-tank volumes of oil. In SI units, the formation-volume factor is a dimensionless quantity, but in oilfield units it is usually reported in units reservoir barrels per stock-tank barrels [bbl/STB], where 1 bbl equals 0.1589873 m^3 . Some authors prefer the reciprocal quantity, $b_o = 1/B_o$, which describes the factor by which one reservoir volume of oil shrinks when brought to surface conditions. MRST uses shrinkage factors rater than formation-volume factors.

B_g – The gas formation-volume factor is defined similarly, as the volume of gaseous phase that must be recovered from the reservoir to produce a unit volume of gas at the surface, i.e., $B_g = V_g/V_{gs}$. It is common to express B_g in units reservoir barrels per standard cubic feet [bbl/scf]. In MRST, we use the reciprocal expansion factor $b_g = 1/B_g$.

R_s – The solution gas-oil ratio is the volume of gas at standard conditions that dissolves into a unit stock-tank volume of oil at reservoir conditions, and is usually reported in units standard cubic feet per stock-tank barrel [scf/STB]. A reservoir volume V with porosity ϕ and oil saturation S_o thus contains $\phi VS_o R_s/B_o$ surface volumes of free gas.

This description assumes that the reservoir is in instantaneous phase equilibrium. In addition to these three parameters, we also need the bubblepoint p_b of the oil, a similar formation-volume factor for water, as well as the dynamic viscosities of the gaseous, oleic, and aqueous phase as function of pressure (and temperature). To model volatile-oil, retrograde-dewpoint, and wet-gas reservoirs, one also need to model the amount of vaporized oil present in the gaseous phase:

R_v – The vaporized oil-gas ratio (or volatile oil-gas ratio) is defined as the stock-tank volume of condensate oil per unit volume of free gas at surface conditions produced from the gaseous phase at reservoir conditions.

The formation-volume factors and gas-oil and oil-gas ratios all depend on reservoir pressure and temperature and are determined by laboratory experiments (or simulations) that reproduce the sequence of differential (and flash) liberations that most likely take place during production of the reservoir. Since reservoirs are almost thermally inert, it is usually sufficient to express these quantities as function of pressure and not temperature.

In the following, we will discuss various fluid properties and relationships in some more detail and in most cases try to mention what are plausible ranges of their values. Readers interested in a more in-depth discussion should consult a text book on reservoir fluids like e.g., [172, 260] or www.petrowiki.org.

Equation-of-state: fluid densities

Using the quantities defined above, we can write the following equations-of-state that relate the densities of the aqueous, oleic, and gaseous phases (denoted by capital letters W, O, G) at reservoir conditions to the densities of the oil, water, and gas pseudo-components (denoted by o, w, g) at surface conditions

$$\rho_W = \frac{\rho_{ws}}{B_w} = b_w \rho_{ws} \quad (11.3)$$

$$\rho_O = \frac{\rho_{os} + R_s \rho_{gs}}{B_o} = b_o (\rho_{os} + R_s \rho_{gs}) \quad (11.4)$$

$$\rho_G = \frac{\rho_{gs} + R_v \rho_{os}}{B_g} = b_g (\rho_{gs} + R_v \rho_{os}) \quad (11.5)$$

Water density is almost invariant to pressure changes. The oleic phase behaves very differently above and below the bubblepoint. Above, the oleic phase behaves like a standard fluid in the sense that the density increases with increasing pressure. Below, gas dissolution will in most cases dominate compression effects, so that density decreases with increasing pressure as a result of dissolved gas that increases the fraction of lighter hydrocarbon components. The gaseous phase is strongly affected by fluid compression so that density increases with pressure.

API and specific gravity

In our discussion of phase behavior, we repeatedly talked about light and heavy hydrocarbon components in the mixture. In petroleum literature, it is common to use the American Petroleum Institute gravity as a measure of how heavy or light a petroleum liquid is. The API gravity is defined as,

$$\gamma_{API} = \frac{141.5}{\gamma_\ell} - 131.5, \quad \gamma_\ell = \frac{\rho_\ell}{\rho_w}. \quad (11.6)$$

where γ_ℓ is the specific gravity of liquid ℓ to water, i.e., the ratio of the density of liquid to the density of water. Bubblepoint pressure p_b , oil viscosity μ_o , and the solution gas-oil ratio R_s , all depend strongly on oil gravity. It also has an indirect effect on oil compressibility and oil formation-volume factors, since these depend on R_s .

The API gravity is defined as degrees on a hydrometer ($^{\circ}$ API) and is defined so that any liquid having API gravity greater than 10° is lighter and floats on water, whereas liquids with values less than 10° are heavier and sink. Using this scale, crude oil is commonly classified as follows:

- light oil: API gravity higher than 31.1° , i.e., less than 870 kg/m^3
- medium oil: API gravity between 22.3 and 31.1° , i.e., 870 to 920 kg/m^3
- heavy oil: API gravity below 22.3° , i.e., 920 to 1000 kg/m^3
- extra heavy oil: API gravity below 10.0° , i.e., greater than 1000 kg/m^3

Typical API gravity values are also reported in the overview of fluid characteristics for different reservoirs in Table 11.1.

For gas, the specific gravity is measured relative to air and is defined as

$$\gamma_g = \frac{\rho_g}{\rho_{air}} = \frac{M_g}{M_{air}},$$

where M denotes the molar mass and $M_{air} = 28.94 \text{ kg/kmol}$. Specific gas gravity is one of the parameters that determine bubblepoint pressure, viscosity and compressibility of gas, and the solution gas-oil ratio. Typical values range from 0.55 for dry gas to approximately 1.5 for wet sour gas. Petroleum gases have gravity approximately equal 0.65.

Reservoir water is saline and hence has a specific gravity greater than unity, but has no impact on compressibility, formation-volume factor, and viscosity.

Compressibility and formation-volume factor of dry gas

You may recall from Chapter 4.2, the equation-of-state of an ideal gas is given as $pV = nRT$, where n is the number of moles of gas and R is the universal gas constant. The literature is full of equations-of-state for real gases, but in petroleum engineering, it is common to use a so-called *compressibility equation-of-state* to describe dry gas

$$pV = ZnRT, \quad Z = \frac{V_{\text{real}}}{V_{\text{ideal}}}, \quad (11.7)$$

where the *compressibility factor* or *gas deviation factor* Z measures how much the volume of a real gas deviates from the volume of an ideal gas at the same temperature. The gas compressibility factor typically varies between 0.8 and 1.2. For constant temperature and moderate pressures, the gas molecules are close enough to exert an attraction on each other that causes the actual gas volume to be less than that of an ideal gas. As p approaches 0, the molecules move more apart and the gas behaves like an ideal gas. At higher pressures,

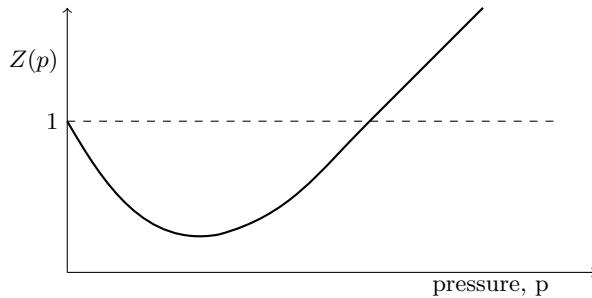


Fig. 11.18. Characteristic shape of gas compressibility factor Z as function of pressure for fixed temperature.

repulsive forces will tend to force molecules more apart and the real gas therefore has a higher volume than an ideal gas. The same type of equation-of-state can be used also for wet gas and mixtures, if Z is replaced by a pseudo function that depends on the composition of the mixture. For volatile oils and retrograde-condensate gases it is more common to use a more advanced equation-of-state like one of the cubic equations discussed in Chapter 4.2.

For a dry gas, the formation-volume factor can be derived from the equation-of-state (11.7),

$$B_g = \frac{p_s}{Z_s T_s} \frac{ZT}{p}. \quad (11.8)$$

where subscript s refers to standard conditions. This varies throughout the world, and can be e.g., 14.65 psia and 60° F or 1 atm and 25° C.

The isothermal compressibility of the gas is defined as the fractional change of volume with respect to pressure for constant temperature,

$$c_g = -\frac{1}{V} \left(\frac{\partial V}{\partial p} \right)_T = -\frac{1}{B_g} \left(\frac{\partial B_g}{\partial p} \right)_T \quad (11.9)$$

Inserting the compressible equation-of-state, it follows that

$$c_g = -\frac{1}{V} \left[\frac{nRT}{p} \left(\frac{\partial Z}{\partial p} \right)_T - \frac{ZnRT}{p^2} \right] = \frac{1}{p} - \frac{1}{Z} \left(\frac{\partial Z}{\partial p} \right)_T.$$

At low pressures, $\partial Z/\partial p$ is negative and hence the compressibility of a real gas is larger than that of an ideal gas. At high pressures, $\partial Z/\partial p$ is positive and hence the compressibility is lower than for an ideal gas.

Gas viscosity

Dynamic gas viscosities typically range from 0.01 to 0.03 cP and are generally difficult to determine experimentally. It is therefore common to use certain correlations to determine values. At low pressures, gas molecules are far apart and can easily move past each other. As pressure increases the molecules are

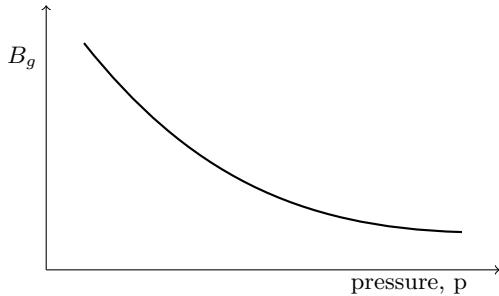


Fig. 11.19. Characteristic shape of gas formation-volume factor as function of pressure for fixed temperature and chemical composition.

forced closer to each other and hence the viscosity of the gas increases. In MRST, gas viscosities are either given as simple analytical expressions (as we have seen so far) or as tabulated input read from an ECLIPSE simulation deck (as we will see later).

Bubblepoint pressure

Bubblepoint pressure is one of the key parameters in the phase behavior of reservoir fluids since it marks the limit between the liquid and two-phase liquid–vapor region for hydrocarbon fluids. In the previous section we outlined briefly how bubblepoint pressure can be measured when studying fluid expansion as function of pressure and temperature. The literature is full of correlations that can be used to determine the bubblepoint pressure (see e.g., [petrowiki.org](#) for an overview), all on the form $p_b = f(T, \gamma_{API}, \gamma_g, R_s)$. None of these are implemented in MRST. Instead, the software follows the convention from leading commercial simulators and require tabulated B and R factors, in which p_b is only given implicitly.

Solution gas-oil ratio and vaporized oil-gas ratio

When discussing dissolved gas, we are in reality referring to light hydrocarbon molecules that appear in gaseous phase at the surface and in liquid oleic phase at reservoir conditions. For a given temperature, the quantity of light molecules present in the oleic phase is limited only by the pressure and the number of light molecules present. For pressures below the bubblepoint, the oleic phase will release some free gas if the pressure is lowered slightly, and hence we say that the phase is *saturated*. Above the bubblepoint, the oleic phase is able to dissolve more gas than what is available, and is said to be *undersaturated*. Altogether, this means that a plot of the quantity of dissolved gas in the oleic phase as function of pressure will therefore increase linearly up to the bubblepoint and then remain constant. The vaporized oil-gas ratio is only defined for pressures below the bubblepoint.

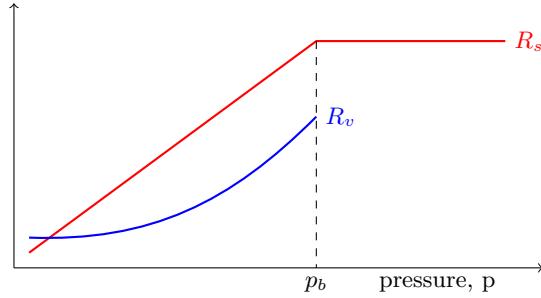


Fig. 11.20. Characteristic shape of the solution gas-oil ratio and vaporized oil-gas ratio as function of pressure for fixed temperature and chemical composition.

Oil formation-volume factor

Most measures of fluid production has traditionally taken place at the surface. It has therefore been customary to introduce the formation volume-factor to relate surface volumes measured at surface conditions to volumes at reservoir conditions. The oil formation-volume factor is affected by *three* mechanisms: The most important is the liberation of gas when bringing liquid oil from the reservoir to the surface, which will cause a decrease in the liquid volume, so that the B_o factor always assumes values greater or equal 1; the more gas that is dissolved, the larger the B_o factor. At reservoir conditions, increased pressure will introduce two additional effect, fluid compression and swelling caused by dissolution of gas, of which the latter is dominant. Below the bubblepoint, the formation-volume factor will therefore increase with increasing pressures. Above the bubblepoint, there is no more gas to be dissolved, and hence fluid compression will cause a gradual decrease in B_o , see Figure 11.21.

The compressibility of oil is defined in the same way as for gas. For undersaturated oil, we therefore have that

$$c_o = -\frac{1}{B_o} \frac{\partial B_o}{\partial p}, \quad B_o = B_{ob} \exp(c_o(p_b - p)), \quad p > p_b \quad (11.10)$$

where B_{ob} is the oil formation-volume factor at the bubblepoint.

Total formation-volume factor

To understand the behavior for saturated oil, we introduce the total or two-phase formation-volume factor, defined as the ratio between the total hydrocarbon volume at reservoir conditions and the volume of oil or gas at standard conditions. For simplicity, let us first consider the case of a black oil with a surface oil volume V_o , i.e., a system with dissolved gas but no vaporized oil. At the bubblepoint, the reservoir fluids inside the reservoir are found as a single oleic phase having a volume $V_o B_{ob}$ and containing $V_o R_{sb}$ volumes of dissolved oil. If we lower the pressure slightly, the system will enter the two-phase region,

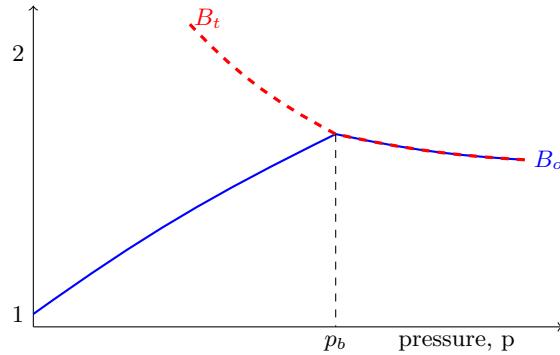


Fig. 11.21. Characteristic shape of the oil formation-volume factor and the total formation-volume factor as function of pressure for fixed temperature and chemical composition.

so that the oleic phase will liberate an amount of gas that at the surface would amount to a volume $V_o(R_{sb} - R_s)$. At reservoir conditions, this corresponds to a volume $B_g(R_{sb} - R_s)$ of free gas. Hence, the total formation-volume factor with respect to oil reads,

$$B_{to} = B_o + B_g(R_{sb} - R_s).$$

Because of the expanding gas, the B_{to} function can have significantly larger values at lower pressures than at the bubblepoint.

For a retrograde gas condensate reservoir, we can start at the dewpoint and carry out a similar argument to obtain

$$B_{tg} = B_g + B_o(R_{vd} - R_v),$$

where R_{vd} is the vaporized oil-gas ratio at the dewpoint.

For volatile oils, things get a bit more complicated as Figure 11.22 illustrates. If no water is present, the pore volume V_ϕ is filled by the oleic and gaseous phase that together contain an amount V_o of oil and V_g of gas. Let V_{of} be the stock-tank volume of free oil and V_{gf} the surface volume of free gas. Then, we have

$$V_\phi = V_{of}B_o + V_{gf}B_g.$$

For pressures below the bubblepoint, oil and gas will appear in both the oleic and gaseous phase, and hence $V_o = V_{of} + V_{gf}R_v$ and $V_g = V_{gf} + V_{of}R_s$. Combining these two and solving for V_{gf} gives $V_{gf} = (V_g - V_oR_s)/(1 - R_vR_s)$. Inserting into the equation above and simplifying, we have

$$V_\phi = \frac{(B_o - R_s B_g)V_o + (B_g - R_v B_o)V_g}{1 - R_s R_v}. \quad (11.11)$$

Just above the bubblepoint, the hydrocarbon is in liquid phase and hence all gas is resolved in the oil, i.e., $V_g = V_o R_{sb}$. By definition, $V_\phi = B_{to} V_o$. Inserting

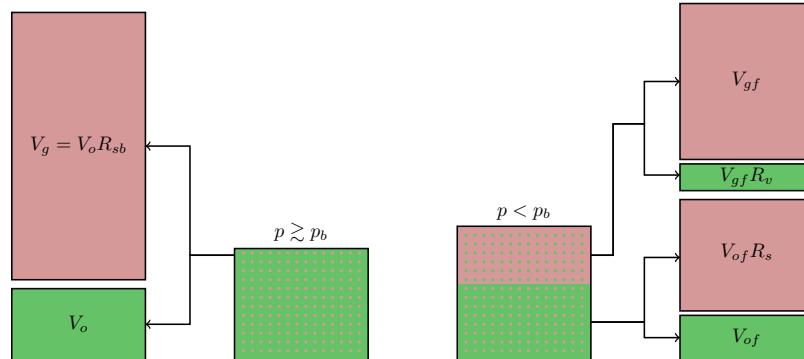


Fig. 11.22. Illustration of the volumes involved in a volatile oil model.

into the above equation, dividing by V_o , and reorganizing terms, we obtain the following expression for the following two-phase formation-volume factor

$$B_{to} = \begin{cases} B_o, & p \geq p_b, \\ \frac{B_o(1-R_{sb}R_v) + B_g(R_{sb}-R_s)}{1-R_vR_s}, & p < p_b. \end{cases} \quad (11.12)$$

The two-phase formation-volume factor for retrograde condensate gas can be derived similarly as

$$B_{tg} = \begin{cases} B_g, & p \geq p_d, \\ \frac{B_g(1-R_{vd}R_s) + B_o(R_{vd}-R_v)}{1-R_vR_s}, & p < p_d. \end{cases} \quad (11.13)$$

Isothermal oil compressibility

Isothermal oil compressibility is defined as the relative change in oil volume per change in pressure. From the discussion above, it follows that the oil compressibility will have different behavior above and below the bubblepoint. For an undersaturated oleic phase above the bubblepoint, the compressibility is defined in the same way as for a gas, i.e., $c_o = (-1/B_o)(\partial B_o/\partial p)_T$, and is a primary drive mechanism. At the bubblepoint, gas comes out of solution and causes a sharp increase in compressibility. Below the bubblepoint, the compressibility becomes a much stronger function of pressure as the oil volume now depends expansion/compression of both the liquid oil and the dissolved gas

$$c_o = -\frac{1}{B_o} \left(\frac{\partial B_o}{\partial p} \right)_T + \frac{1}{B_o} \frac{B_g - B_o R_v}{1 - R_s R_v} \left(\frac{\partial R_s}{\partial p} \right)_T. \quad (11.14)$$

As a drive mechanism, however, oil compressibility is dominated by the compressibility of gas that has come out of solution.

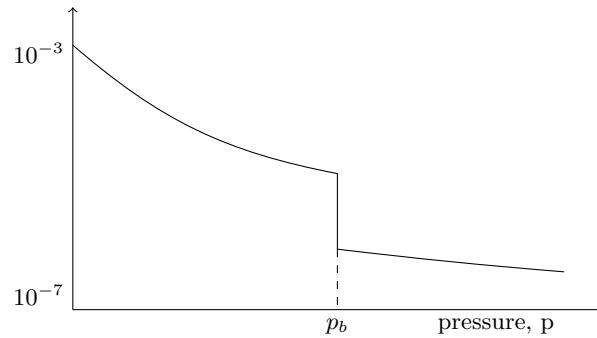


Fig. 11.23. Isothermal oil compressibility as function of pressure for fixed temperature and chemical composition.

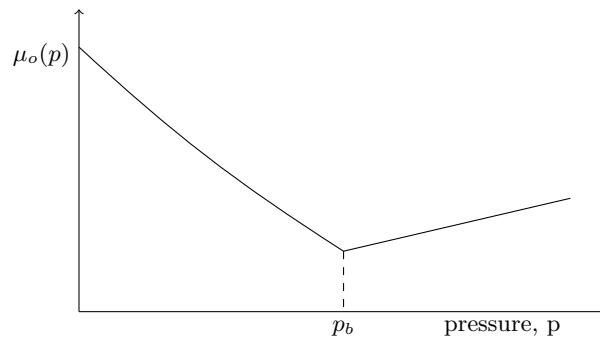


Fig. 11.24. Oil viscosity as function of pressure for fixed temperature and chemical composition.

Oil viscosity

We have already seen in Chapter 7.3 that the viscosity of a fluid will be affected by both temperature and pressure, with temperature being the most dominant for a fluid of constant composition. Viscosity is also related directly to the type and size of molecules the fluid is made up of. Above the bubblepoint, the oleic phase has a constant chemical composition and the lower the pressure is, the easier the molecules can move past each other, since they are further apart. Hence, the viscosity increases almost linearly with pressure. Below the bubblepoint, the oleic phase changes chemical composition as more lighter molecules are removed from the liquid when gas is liberated. The fraction of larger molecules with complex shapes will gradually become larger, causing the viscosity to increase. One order of magnitude increase in viscosity from bubblepoint to low reservoir pressure is not uncommon, which will largely reduce the mobility of the remaining oil.

Water properties

The aqueous phase is usually a brine that contains a number of different dissolved salts and hence has specific density greater than unity. Water may also contain certain amounts of dissolved gas, but this is often ignored in standard simulation models since the solubility factor in water is orders of magnitude lower than in the liquid oleic phase.

Water viscosity decreases with increasing temperature and increases with increasing pressure, but is generally a very weak function of pressure. Typical values at reservoir conditions range from 0.5 to 1 cP. Water is only slightly compressible at reservoir conditions, with values in the range from $1 \cdot 10^{-6}$ to $9 \cdot 10^{-6}$ psi⁻¹, which implies that the water formation-volume factor is usually close to unity.

11.4 Phase behavior in ECLIPSE input decks

Use of tabulated data from the `PROPS` section of an ECLIPSE input file is the primary input mechanism to set up simulation models for the black-oil type simulators implemented in the `ad-blackoil` (and `ad-eor`) modules. As we saw earlier on page 375, functionality for reading these input files, parsing and processing them to create specific fluid objects is offered by the `deckformat` and `ad-props` modules.

To make the book as self-contained as possible, this section will outline the necessary ECLIPSE keywords and show a few examples of specific phase behavior for the SPE 1, SPE 3, and SPE 9 benchmark cases. For a more comprehensive discussion of the ECLIPSE input format, the reader should consult the lecture notes of Pettersen [213] or the ECLIPSE [226, 228] or OPM Flow [29] documentation. The following presentation is inspired these sources.

Dead oil

Dead oils are assumed to always stay above the bubble point and hence behave like a standard fluid that is compressed and becomes more viscous at higher pressures. Most dead oils contain a constant amount of dissolved gas that is liberated as the oleic phase is brought to the surface. There are two different ways to describe dead oil in an ECLIPSE input deck:

`PVDO` – specifies the properties of an undersaturated oil above the bubble-point in terms of a table of formation-volume factors and viscosities versus monotonically increasing pressure values. The table is attached to specific (groups of) cells, and the keyword can contain multiple tables if the reservoir contains several distinct oil. The number of tables is given by the `NTPVT` element of the `TABDIMS` keyword from the `RUNSPEC` section. The constant amount of dissolved oil and the associated bubblepoint pressure

should be specified using the `RSCONST` keyword. The following example is taken from [213]:

```
PVDO
-- Po    Bo      mu_o
  27.2   1.012   1.160
  81.6   1.004   1.164
 136.0   0.996   1.167
 190.5   0.988   1.172
 245.0   0.9802  1.177
 300.0   0.9724  1.181
 353.0   0.9646  1.185 /
RSCONST
-- Rs    Pbp
  180   230 /
```

Here, the pressures `Po` and `Pbp` are given in units bar, the formation-volume factor `Bo` and the gas-oil ratio `Rs` are dimensionless, and the viscosity `mu_o` has unit cP.

`PVCDO` – describes an oleic phase with constant compressibility by specifying the five constants in the formulas

$$B_o(p) = B_{or} e^{-c_o(p-p_r)}, \quad (B_o \mu_o)(p) = B_{or} \mu_{or} e^{-(c_o - c_\mu)(p-p_r)}.$$

Here, p_r is reference pressure, B_{or} reference formation-volume factor, c_o compressibility, μ_{or} reference viscosity, c_μ and 'viscosibility'. In addition, one must specify `RCONST` and a maximum pressure `PMAX`. The following example is taken from [228] (with pressures given in units psi):

```
PVCDO
-- P    Bo      Co      mu0     Cmu
  2500  1.260   6E-6   0.5     1E-6 /
RSCONST
-- GOR    Pb
  0.656   2500 /
PMAX
  4500 /
```

Live oil

Undersaturated oil has a well-defined bubblepoint that can be determined from the temperature and the oil's chemical composition. Saturated oils, on the other hand, change chemical composition as gas is dissolved or liberated. To model the phase behavior of a live oil, one needs to describe both how the bubblepoint changes with changing composition of the oil, as well as the properties in the undersaturated region for each unique composition. In a black-oil model, the fluid composition of the oleic phase is represented in terms of the solution gas-oil ratio, and all we need to discriminate saturated from undersaturated states is a relationship between R_s and the bubblepoint

p_b . In ECLIPSE, this is represented by tabulating $p_b(R_s)$. For each value of $p_b(R_s)$, the dead-oil behavior in the undersaturated region can be prescribed in the same way as described above. Hence, ECLIPSE offers two keywords for describing live oils:

PVT0 – consists of NTPVT tables that each give the properties of above and below the bubblepoint. Each table consists from 2 to NRPVT records that specify PVT data for given R_s values. The records consist of at least four numbers specifying R_s , bubblepoint pressure p_b , formation-volume factor B_{ob} for saturated oil at the bubble point, and viscosity μ_{ob} at the bubblepoint. However, each record can also be composed of from 2 to NPPVT rows of data organized in three columns that describe oil pressure (increasing values), formation-volume factor (decreasing values), and viscosity for undersaturated oil for the specified value of R_s . These extra data *must* be specified for the maximum R_s value inside each table but can optionally be given for any R_s value. A specific example of this input format is presented for the SPE 9 model on page 407.

Notice that ECLIPSE assumes interpolation of the reciprocal values of B_o and ($\mu_o B_o$) between data points.

PVCO – specifies properties above and below the bubblepoint in compressibility form; that is, instead of specifying the undersaturated behavior in terms of a tabular, it is described as for the PVCDO keyword. The data thus consists of NTPV tables that each may have up to NRPVT rows. The following example is from Eclipse [228]:

```

PMAX
  6000 /
PVCO
--Pb    Rs     Bo      Vo      Co      Cvo
  1214.7  0.137  1.17200  1.97000  1E-5  0
  1414.7  0.195  1.20000  1.55600  1*    0
  1614.7  0.241  1.22100  1.39700  1*    0
  1814.7  0.288  1.24200  1.28000  1*    0
  2214.7  0.375  1.27800  1.09500  1*    0
  2614.7  0.465  1.32000  0.96700  1*    0
  3014.7  0.558  1.36000  0.84800  1*    0
  3414.7  0.661  1.40200  0.76200  1*    0
  3814.7  0.770  1.44700  0.69100  1*    0
/

```

Notice, in particular the use of 1* to indicate a default value that will be computed by linear interpolation. Since the table only gives one value in the first row, this value will be copied to all the next rows.

Dry gas

Dry gases are found either below the dewpoint or (well) to the left of the critical point. Their phase behavior is equivalent to that of dead oils and can

be described as a tabular of formation-volume factor and viscosity as function of pressure or Z -factor and viscosity as function of pressure:

PVDG – specifies NTPVT tables that each consist of three columns and up to NPPVT rows giving gas pressure (increasing monotonically downwards), gas formation factor, and gas viscosity. A specific example of this input format is presented for the SPE 9 model on the next page.

PVZG – specifies properties using Z -factors. The data consists of NTPV tables that each consists of two records. The first gives the reference temperature used to convert Z -values into formation-volume factors (see (11.8) on page 397), and the second consists of up to 2 to NPPVT rows giving gas pressure, Z factor, and gas viscosity. Pressures and Z values should increase monotonically down the table. The lowest pressure value should correspond to the dewpoint. The gas may contain a constant amount of vaporized oil, specified by the RVCONST keyword, which also gives the dew-point pressure.

Wet gas

Wet or rich gases with vaporized (condensate) oil are similar to live oils in the sense that the phase behavior is different for saturated and undersaturated states. The composition of the wet gas is represented by vaporized oil-gas ratio R_v . To discriminate between saturated and undersaturated states, the maximum amount of vaporized oil is tabulated as function of pressure, i.e., $R_v(p_g)$. The corresponding ECLIPSE keyword is:

PVTG – describes properties of wet gas with vaporized oil. The format is similar to the **PVTO** keyword used to describe live oil, and consists of from 2 to NRPVT records that specify PVT data (R_v , B_g , and μ_g) at the saturated state for given p values. Each record can also be composed of from 2 to NPPVT rows of three-column data that specify R_v , B_g , and μ_g for undersaturated states. These extra data *must* be specified for the highest pressure data in each of the NTPVT tables.

An example of this input format is shown on page 410 when discussing the PVT behavior of the SPE 3 benchmark.

Miscellaneous properties

Water properties are described with the keyword **PVTW**, which follows the same format as the **PVCDO** keyword for dead oil. In addition to the PVT behavior, one also needs to prescribe the density or specific gravity of each fluid at surface condition, which can be defined by the keywords **DENSITY** or **GRAVITY**. The **PROPS** section of the input deck should also describe how the pore volume depends on pressure, either by the **ROCK** keyword that specifies a reference pressure and rock compressibility, or by the **ROCKTAB** keyword that tabulates porosity (and permeability) multipliers as function of pressure.

Example: SPE 1 and SPE 9 – live oil with dry gas

To exemplify live oil and dry gas models, we revisit the SPE 1 and SPE 9 benchmarks discussed on page 378. We will not repeat the statements required to load these model, but simply assume that we have created a processed `deck` structure with the necessary input data and created a fluid object `f`. The SPE 9 benchmark describes a live oil with dry gas specified as follows (the setup for SPE 1 is similar):

-- LIVE OIL (WITH DISSOLVED GAS)				-- DRY GAS (NO VAPORIZED OIL)		
PVTO				PVDG		
-- Rs Ppub Bo Vo				-- Pg Bg Vg		
.0 14.7 1.0000 1.20 /				14.7 178.08 .0125		
.165 400. 1.0120 1.17 /				400. 5.4777 .0130		
.335 800. 1.0255 1.14 /				800. 2.7392 .0135		
.500 1200. 1.0380 1.11 /				1200. 1.8198 .0140		
.665 1600. 1.0510 1.08 /				1600. 1.3648 .0145		
.828 2000. 1.0630 1.06 /				2000. 1.0957 .0150		
.985 2400. 1.0750 1.03 /				2400. 0.9099 .0155		
1.130 2800. 1.0870 1.00 /				2800. 0.7799 .0160		
1.270 3200. 1.0985 .98 /				3200. 0.6871 .0165		
1.390 3600. 1.1100 .95 /				3600. 0.6035 .0170		
1.500 4000. 1.1200 .94				4000. 0.5432 .0175 /		
5000. 1.1189 .94 /						
/						
PVTW				DENSITY		
--Pref Bw Comp Vw Cv				-- Oil Water Gas		
3600. 1.0034 1.0E-6 0.96 0.0 /				44.98 63.01 0.0702 /		

The model is specified in field units, so that densities are given as lb/ft³ and not kg/m³ and pressure values are given in unit psi. For the live oil, we see that undersaturated behavior is described for the largest R_s value. Once the units have been converted to the SI system and the input data have been processed to construct a fluid object, the phase behavior of the oil is represented by the following functions and constants:

```
B0: @(po,rs,flag,varargin) B0(po,rs,pvto,flag,reg,varargin{:})
b0: @(po,rs,flag,varargin) b0(po,rs,pvto,flag,reg,varargin{:})
mu0: @(po,rs,flag,varargin) mu0(po,rs,pvto,flag,reg,varargin{:})
rsSat: @(po,varargin) rsSat(po,pvto,reg,varargin{:})
rho0S: 720.5105
```

MRST has functions for computing both formation-volume factors and the reciprocal expansion/shrinkage factors for completeness. We also note that R_s is called `rsSat` rather than `Rs`. The reason is that dissolved gas will be an unknown in our system of flow equations and hence we change the name of the *parameter* R_s that describes the maximum dissolved gas amount for a given pressure to avoid confusion.

Looking at the functions in more detail, we see that the `B0` function takes three arguments: the oil pressure, the amount of dissolved oil, and a flag that indicates whether a given element in `po` and `rs` represents a saturated or an undersaturated state. In the underlying functions, `pvto` is an object that holds the data of the `PVTO` keywords, whereas `reg` represents region information that

picks the right table, should the model have multiple oils. The implementation of the `B0`, `b0`, `mu0`, and `rSat` functions can be found in the file `assignPVT0.m` and consists of calls to two utility functions that extract region information and then using this to interpolate the correct data tables. For saturated oil, we use a simple linear interpolation between the data points, whereas the interpolation for undersaturated oil is set up to preserve compressibility and viscosibility if only one undersaturated date curve is given, or to interpolate compressibility and viscosibility if multiple curves are given. I will not go through the interpolation functions in detail, which have been optimized for speed and hence are only easy to understand if you are a black-belt MATLAB programmer.

Instead, we can plot the various relationships (the necessary code can be found in the `showSPEfluids` script in the `book` module). We start by extracting the input data from the `deck` structure:

```
pvto = deck.PROPS.PVTO{1};  
rsd = pvto.key([1:end end]);  
pbp = pvto.data([pvto.pos(1:end-1)' end],1);  
Bod = pvto.data([pvto.pos(1:end-1)' end],2);  
mu0d = pvto.data([pvto.pos(1:end-1)' end],3);
```

Here, we have only extracted the data points representing saturated states as well as the undersaturated data for the maximum R_s value, which is the minimal data requirement and coincides with what is given in the input file in our case. To simplify subsequent interpolation, the input processing in MRST adds an extra data point inside the undersaturated region for each data point, but these are ignored in the above data extraction. R_s values for *all* data points represented in the `deck` structure can be constructed as follows:

```
rsd = pvto.key(rldecode((1:numel(pvto.key))',diff(pvto.pos),1));
```

To evaluate the fluid model, we make a mesh of pressure and dissolved gas states, and for each state, we determine whether it represents a saturated or undersaturated state by comparing with the interpolated R_s value for the corresponding pressure:

```
[RsMax,pMax] = deal(max(rsd), max(pbp));  
[rs,p] = meshgrid(linspace(10,RsMax-10,M), linspace(0,pMax,N));  
Rs = reshape(f.rsSat(p(:)), N, M);  
isSat = rs >= Rs;  
rs(isSat) = Rs(isSat);  
Bo = reshape(f.B0 (p(:, rs(:, isSat(:))),N,M);  
mu0 = reshape(f.mu0(p(:, rs(:, isSat(:))),N,M);
```

Figure 11.25 shows plots of R_s as function of pressure and B_o and μ_o as function of pressure and amount of dissolved gas for the SPE 1 and SPE 9 fluid models. Whereas the SPE 1 benchmark describes a gas-injection scenario, the SPE 9 benchmark describes a case with water injection. As we can see,

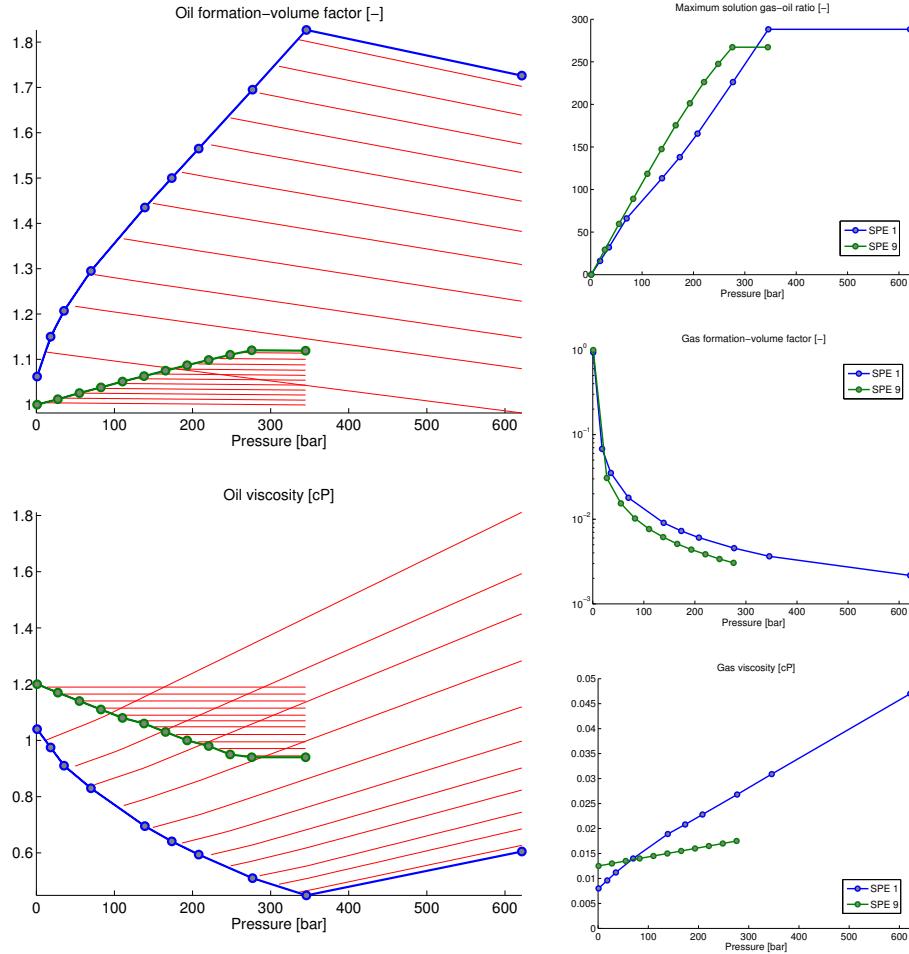


Fig. 11.25. Description of phase behavior for the live-oil and dry-gas models in the SPE 1 and SPE 9 benchmarks, shown as blue and green lines, respectively. In the B_o and μ_o plots, the red lines correspond to undersaturated states.

there is a pronounced difference in the phase properties of the two models. Whereas the SPE 1 oil shows strong compressibility and viscosibility effects both for the saturated and undersaturated states, the undersaturated SPE 9 oil is almost incompressible and has constant viscosity. In part, this is also indicated by the surface densities of oil and gas, which are 786.5 kg/m³ and 0.9698 kg/m³ for SPE 1 and 720.5 kg/m³ and 1.1245 kg/m³ for SPE 9, which means that SPE 1 has a larger fraction of both lighter and heavier hydrocarbon components.

For completeness, let us also look at the gaseous and aqueous phases. The behavior of dry gas is represented by the following functions and constants (here with values for SPE 9):

```
BG: @(pg,varargin) ireg(TBG,pg,varargin{:})
bG: @(pg,varargin) ireg(TbG,pg,varargin{:})
muG: @(pg,varargin) ireg(TmuG,pg,varargin{:})
rhoGS: 1.1245
```

Here, `TBG`, `TbG`, and `TmuG` are data objects containing the formation-volume factor, its reciprocal expansion factor, and the viscosity from the PVDG keyword, and `ireg` is a shorthand for the `interpReg` function we encountered when discussing computation of three-phase relative permeabilities on page 377. The aqueous phase is represented as a weakly compressible fluid (here for SPE 9),

```
cW: 1.4504e-10
muWr: 9.6000e-04
BW: @(pw,varargin) BW(pw,pvtw,reg,varargin{:})
bW: @(pw,varargin) bW(pw,pvtw,reg,varargin{:})
muW: @(pw,varargin) muW(pw,pvtw,reg,varargin{:})
rhoWS: 1.0093e+03
```

To be consistent with ECLIPSE, MRST uses a Taylor expansion to approximate the weakly compressible functions

$$B_w(p) = B_{wr} e^{-c_w(p-p_r)} = B_{wr} e^{-\xi} \approx \frac{B_{wr}}{1 + \xi + \frac{1}{2}\xi^2}.$$

The interested reader should also consult the tutorial `fluidInspectionExample` from the `ad-core` module, which introduces a graphical interface for fluid models made in the object-oriented AD-OO framework that produces many of the same plots as shown above and in Section 11.2. Creating models and launching the GUI can be done quite compactly:

```
[G, rock, fluid, deck] = setupSPE9();
spe9 = selectModelFromDeck(G, rock, fluid, deck);
inspectFluidModel(spe9)
```

Example: SPE 3 – rich gas with retrograde condensation

The SPE 3 benchmark was designed to compare the ability of compositional simulators to match PVT data for a problem with gas cycling in a rich-gas reservoir with retrograde condensation. The input deck distributed with MRST is a translation of the problem into a black-oil framework, with which one can only hope to describe the pertinent flow physics to a limited extent. The resulting model consists of a dead oil, described with the `PVD0` keyword, and a rich gas described by the `PVTG` keyword. Here, we only show excerpts of the specification of the rich gas:

-- 'Gas Pressure'	'Gas OGR'	'Gas FVF'	'Gas Visc'
-- Units: psia	stb /Mscf	rb /Mscf	cp
PVTG			
1214.7000	0.0013130	2.2799	0.0149
	0	2.2815	0.01488/
1814.7000	0.00353	1.4401	0.01791
	0.001313	1.4429	0.01782
	0	1.4445	0.01735 /
2414.7000	0.01102	1.0438	0.02328
:			
3449.3322	0.0642	0.7783	0.0403
	0.0612	0.7777	0.0395
	0.0567	0.7769	0.03892
	0.0454	0.7747	0.03748
	0.0331	0.7723	0.03594
	0.01102	0.7681	0.03325
	0.00353	0.7666	0.03236
	0.001313	0.7662	0.0321
	0	0.7660	0.03194 /
/			

The maximum amount of vaporized oil increases as function of pressure, and the input table uses an increasing number of data points within each record, starting with two points for the first pressure value and ending with nine values for the highest pressure value. The data points are quite unevenly scattered in (p_g, R_v) space, as can be seen from the plot of R_v and μ_g versus p_g in Figure 11.26. One should therefore be wary of potential numerical inaccuracies in the numerical interpolation. (The full code used to generate the plots in the figure are given in the `showSPEfluids` script in the `book` module.)

Looking at the qualitative behavior of the gas properties shown in Figure 11.26, we see that as pressure increases, more oil can be vaporized into the gas and this will increase the viscosity of the gaseous phase. The data points along the vertical lines traverse the phase envelope for fixed pressure, from the blue line corresponding to a saturated state with full vaporization (i.e., maximal supplied R_v value) to the red line representing the dewpoint, at which the vapor phase contains no vaporized oil ($R_v = 0$). Vaporization of oil also affects the gas formation-volume factor, but since the amount of vaporized oil is so small, the resulting variations in volume across the phase envelope for a fixed pressure are so small that they are only visible when zooming in on the plot. Interestingly, we observe that B_g decreases with increasing vaporization for pressures below 220 bar, but *increases* with increasing vaporization for pressures above 220 bar. However, the density of the gaseous phase increases monotonically with increasing oil vaporization, as one would expect physically.

The figure does not report properties of the dead oil, but these behave as expected, with B_o decreasing and μ_o increasing with increasing pressure.

11.5 The black-oil equations

The flow equations for black-oil reservoir model have already been introduced in Chapter 8.2.3, but for completeness, we will go through them again, com-

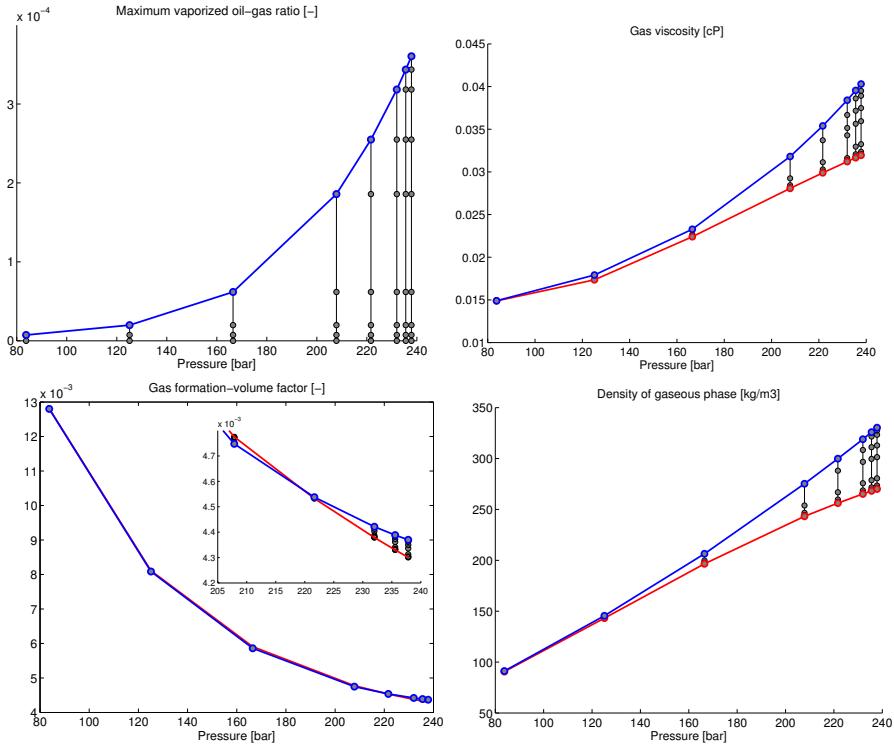


Fig. 11.26. Description of phase behavior for the rich-gas system of the SPE 3 benchmark problem. Vertical lines represent transition across the phase envelope for fixed pressure, with the blue line corresponding representing $\max R_v$ and red lines $R_v = 0$.

ponent by component, and discuss the various special cases and present how they can be discretized using discrete differentiation and averaging operators. As should be evident from the discussion above, the basic assumption of the black-oil family of flow models is that the recovery process takes place at isothermal conditions inside the reservoir, so that the PVT behavior can be modelled as a function of pressure only. The resulting models work best when operating in single-phase hydrocarbon regions far from the critical point, but can also be used in two-phase hydrocarbon regions as long as the variation in composition is relatively small.

11.5.1 The aqueous component

The discussion in the previous sections has primarily focused on the gaseous and oleic hydrocarbon phases containing the gas and oil components. Reservoirs will nonetheless always contain aqueous fluids present as connate water within the hydrocarbon-bearing rock layers. The reason is that the reservoir

rock was fully or partially saturated with water before hydrocarbons migrated upwards from a deeper source rock. Since most rocks are fully or partially water-wet, the resident water is not fully displaced by the invading hydrocarbons. Usually, there will also be extensive aquifers underlying the hydrocarbon bearing rocks, and these may contribute pressure support and fluid inflow that should be accounted for in the reservoir model.

As a rule, aqueous reservoir fluids are saline. The most common dissolved salt is NaCl, but the aqueous phase will typically contain a wide range of compositions and concentrations of other inorganic chemical species. These species are not accounted for in basic black-oil type models, which simply assume that the resident brine and injected water is a single, chemically inert component that only is present in the aqueous phase. (The interested reader may consult Trangenstein and Bell [247] for an alternative model, in which gas is allowed to dissolve into the aqueous phase.) The corresponding flow equations read

$$\partial_t(\phi b_w S_w) + \nabla \cdot (b_w \vec{v}_w) - b_w q_w = 0, \quad (11.15)$$

with Darcy's law for the phase velocity

$$\vec{v}_w = -\frac{K k_{rw}}{\mu_w} (\nabla p_w - g \rho_w \nabla z)$$

These equations are discretized exactly in the same way as (11.1) on page 365, except for the obvious modifications resulting from ρ_w having been replaced by b_w . Let us now look at how the discrete flow equations can be implemented in MRST. We have already discussed most of the tools you need. As before, we assume that the fluid properties are collected in a fluid object called `fluid`. For convenience, we have chosen to collect all the discrete differentiation and averaging operators in a special structure in the `ad-core` module, which we here will refer to by the short-hand `s`. We start by computing the necessary fluid properties:

```
bW      = fluid.bW(p0);
rhoW    = bW.*fluid.rhoWS;
rhoWF   = s.faceAvg(rhoW);
mobW    = krW./fluid.muW(p0);
pvMultR = fluid.pvMultR(p0);
```

Here, `faceAvg` is the same as the `avg` operator we have used previously, but has been renamed to specify explicitly that this is an averaging operator that should be used to compute average values at cell interfaces. Notice that the pressure-dependent properties are evaluated using the oil pressure and not the water pressure. In general, there are several choices that can be made for the primary unknowns in the black-oil models. Herein, we have chosen to follow the conventions from ECLIPSE and use oil pressure p_o and gas and water saturations, S_w and S_g , as our primary unknowns. (We will get back to the choice of primary variables later in this section).

The water flux across cell faces will, as before, be computed using single-point upstream-mobility weighting, as discussed previously. To this end, we first need to compute the water pressure in each grid cell,

```
pW = p0
if isfield(fluid, 'pcOW') && ~isempty(sW)
    pW = pW - fluid.pcOW(sW);
end
```

Then, the water flux can be computed from Darcy's law

```
dpW = s.Grad(pW) - rhoWF.*gdz;
upcw = (double(dpW)<=0);
vW = -s.faceUpstr(upcw, mobW).*T.*dpW;
bWvW = s.faceUpstr(upcw, bW).*vW;
```

The function `faceUpstr` implements the upstream operator we previously denoted by `upw`. With this, we have all quantities we need to compute the homogeneous water equation on residual form

```
water = (s.pv/dt).*(pvMult.*bW.*sW - pvMult0.*bW0.*sW0) + s.Div(bWvW);
```

Here, quantities ending with 0 refer to pressure-dependent variables evaluated using the pressure at the beginning of the time step and the vector `s.pv` contains pore volumes in each cell at reference pressure. The code lines above are in essence what is implemented in the flow solvers of the `ad-blackoil` module. However, if you look at the code in e.g., `equationsBlackOil`, you will see that it is more complicated since we in general need to also include (multisegment) well models, source terms, and boundary conditions; account for various user-defined multipliers that modify transmissibilities and mobilities; enable solution of adjoint equations for computing sensitivities; and store computed results.

11.5.2 The oil component

In the basic black-oil model, the oil component is only present in the liquid oleic phase, and we hence get the following flow equation:

$$\partial_t(\phi b_o S_o) + \nabla \cdot (b_o \vec{v}_o) - b_o q_o = 0, \quad (11.16)$$

with Darcy's law for the phase velocity

$$\vec{v}_o = -\frac{Kk_{rw}}{\mu_o} (\nabla p_o - g \rho_o \nabla z).$$

As for the water component, we start by computing the necessary fluid properties. How we compute density, shrinkage factor and viscosity will depend upon whether the oil phase contains dissolved gas or not. If the system contains no dissolved gas, the parameters will only depend on the oil pressure and can be computed as follows:

```

b0 = fluid.b0(p0);
if isfield(fluid, 'B0xmu0')
    mu0 = fluid.B0xmu0(p0).*b0;
else
    mu0 = fluid.mu0(p0);
end
rho0 = b0.*fluid.rho0S;

```

As pointed out above, the convention in ECLIPSE 100 input is that tabulated formation-volume factors and viscosity for oil should not be interpolated independently, but instead be interpolated as their reciprocal product, $1/(\mu_o B_o)$. MRST signifies input data that require this particular interpolation by a special function `B0xmu0` in the fluid object. If this function is not present, the viscosity and the shrinkage factors are computed independently. If oil also contains dissolved gas, the same properties are evaluated as follows:

```

b0 = fluid.b0(p0, rs, isSat);
mu0 = fluid.mu0(p0, rs, isSat);
rho0 = b0.*((rs*fluid.rhoGS + fluid.rho0S));

```

Here, `rs` is a vector holding the gas-oil ratio and `isSat` is a boolean vector that tells whether each element of `p0` and `rs` represents a saturated or an undersaturated state. We will get back to how `rs` is computed in Section 11.5.4. With density, shrinkage factor, and viscosity given, we can proceed to compute the oleic phase flux using upstream-mobility weighting as described for the aqueous phase above. Altogether, this gives us the homogeneous residual equation

```

oil = (s.pv/dt).*( pvMult.*b0.*s0 - pvMult0.*b00.*s00 ) + s.Div(b0v0);

```

If oil is allowed to vaporize into the gaseous phase, the conservation equation for the oil component reads

$$\partial_t [\phi(b_o S_o + b_g R_v S_g)] + \nabla \cdot (b_o \vec{v}_o + b_g R_v \vec{v}_g) - (b_o q_o + b_g R_v q_g) = 0. \quad (11.17)$$

We therefore need to compute the properties and the phase flux for the gaseous phase, before we can compute the homogeneous residual oil equation:

```

oil = (s.pv/dt).*( pvMult.* (b0.* s0 + rv.* bG.* sG) - ...
    pvMult0.* (b00.*s00 + rv0.*bG0.*sG0) ) + s.Div(b0v0 + rvbGvG);

```

Notice in particular that we use single-point upstream-mobility weighting to evaluate the vaporized oil-gas ratio in the term representing flux of oil vaporized in the gas phase. This is completely analogous to how the shrinkage factors for oil and water are evaluated at the interface between two neighboring cells.

11.5.3 The gas component

The gas component will generally be present both in the gaseous and the liquid oleic phases. The corresponding conservation equation reads

$$\partial_t [\phi(b_g S_g + b_o R_s S_o)] + \nabla \cdot (b_g \vec{v}_g + b_o R_s \vec{v}_o) - (b_g q_g + b_o R_s q_o) = 0 \quad (11.18)$$

with Darcy's law for the phase velocity

$$\vec{v}_g = -\frac{K k_{rg}}{\mu_g} (\nabla p_g - g \rho_g \nabla z).$$

Like for oil, the computation of the gas properties differs for systems with and without vaporized oil. Likewise, we also need to compute the phase pressure to determine the phase velocity:

```

if vapoil
    bG   = fluid.bG(p0, rv, isSat);
    muG  = fluid.muG(p0, rv, isSat);
    rhoG = bG.*(rv*fluid.rhoOS + fluid.rhoGS);
else
    bG   = fluid.bG(p0);
    muG  = fluid.muG(p0);
    rhoG = bG.fluid.rhoGS;
end

pG = p0;
if isfield(fluid, 'pc0G') && ~isempty(sG)
    pG = pG + fluid.pc0G(sG);
end

```

With these quantities given, we can proceed as above to compute the remaining terms necessary to evaluate the accumulation and flux terms in the homogeneous residual equation

```

gas = (s.pv/dt).*(
    pvMult.* (bG.* sG + rs.* b0.* s0) - ...
    pvMult0.* (bG0.*sG0 + rs0.*b00.*s00) ) + s.Div(bGvG + rsb0v0);

```

11.5.4 Appearance and disappearance of phases

With all the details presented above, it should be a surmountable task to extend the simple two-phase, dead-oil implementation discussed in Section 11.1 to a similar simplified simulator for black-oil models with dissolved gas in the liquid oleic phase and/or vaporized oil in the gaseous phase as long as each hydrocarbon phase either stays in a saturated state or an unsaturated state throughout the whole simulation. For saturated phases, it is common to use one phase pressure (p_o in MRST) and the water and gas saturation (S_w and S_g) as independent variables since both the solution gas-oil ratio R_s and the

vaporized oil-gas ratio R_v can be computed as function of pressure. If oil is in undersaturated state, there is no free gas available and the flow problem is essentially a two-phase flow problem (assuming that water is present). However, such cases can also be simulated as a three-phase problem if we set R_s as a primary variable. Likewise, we can choose R_v as a primary variable for cases with undersaturated gas.

A bit more intricate details are needed when the hydrocarbon phases switch between saturated and undersaturated states so that phases may appear or disappear during the simulation. As you have seen already, MRST introduces the solution gas-oil ratio rs as an extra variable alongside Sg for cases with dissolved gas. In addition, we need a status flag in each cell that tells which of the two is the primary unknown. When the oil is in a saturated state, Sg is the primary variable and rs is a secondary variable that can be determined from the pressure. If the computed gas saturation in a cell is zero or falls below, we know that the cell has switched from a saturated to an undersaturated state. In cells containing undersaturated oil, there is no free gas available and we solve for rs as a primary variable to track the amount of gas flowing in and out of each cell. Once a new rs value is computed, we need to check whether $rs > \text{fluid.rsSat}(p0)$, which means that we have a transition from an undersaturated to a saturated state and hence need to increase Sg to the correct amount of liberated gas. This may unfortunately introduce instabilities in the nonlinear solution process, and in practice we do not allow states to cross from a saturated/undersaturated state and well into the undersaturation/saturated region, but rather stop the update at the interface, i.e., set rs to $rsMax = \text{rsSat}(p0)$ and $Sg = 0$, and chop the time step to improve robustness of the Newton iteration loop. Cases with transitions between saturated and undersaturated gas are treated analogously. The code below summarizes the logic explained above to choose the primary variable x that represents gas:

```
% Assumption: rs = rsSat for saturated cells, likewise for rv
% status 1 (oil, no gas): x = rs, sg = 0,    rv = rvMax
% status 2 (gas, no oil): x = rv, sg = 1-sw,  rs = rsMax
% status 3 (oil and gas): x = sg, rs = rsMax, rv = rvMax
watOnly      = sw > 1- sqrt(eps);

if ~vapoil, oilPresent = true; else, oilPresent = or(s0 > 0, watOnly); end
if ~disgas, gasPresent = true; else, gasPresent = or(sG > 0, watOnly); end

status = oilPresent + 2*gasPresent;
if ~disgas, st1 = false; else, st1 = status==1; end
if ~vapoil, st2 = false; else, st2 = status==2; end
st3 = status == 3;

x = st1.*rs + st2.*rv + st3.*sG;
```

11.6 Well models

Earlier in the book, we have discussed well models for single-phase flow. Section 4.3.2 introduced the basic inflow performance relationship which states that the flow rate in or out from a well is proportional to the pressure difference between the wellbore and the reservoir, $q = J(p_R - p_{wb})$. This section describes well models for multiphase flow in more detail. In particular, we discuss wells with multiple connections and explain how to handle cross flow and introduce a more comprehensive family of models that use a network of 1D flow models to describe flow in horizontal and multilateral wells.

11.6.1 Inflow performance relationships

The inflow performance relationship for black-oil models is written in terms of volumetric production rates for each fluid phase α at stock-tank conditions

$$q_{\alpha,i} = J_i \lambda_{\alpha,i} (p_i - p_{bh} - H_{wi}). \quad (11.19)$$

Here, J_i is the productivity index (sometimes called connection transmissibility factor or R -factor) for connection number i , $\lambda_{\alpha,i}$ is phase mobility at the connection, p_i is pressure in the perforated grid block, p_{bh} is bottom-hole pressure, and H_{wi} is pressure head from the well's datum point to connection i .

The standard Peaceman-type expression for the productivity index of a vertical well penetrating a Cartesian grid block reads

$$J = \frac{\theta K h}{\ln(r_e/r_w) + S}.$$

Here, $K h$ is effective permeability times net thickness of the connection, r_w is wellbore radius, and r_e is the equivalent radius. We have previously discussed expressions for K and r_e for various special cases in Section 4.3.2. The parameter θ gives the angle the wellbore is connected to the reservoir block, with $\theta = 2\pi$ when the well is perforated in the center of the cell, $\theta = \pi$ when the well is perforated along a cell face, and $\theta = \pi/2$ when the well is perforated along a corner edge; see Figure 11.27. The skin factor S accounts for the difference between ideal pressure drawdown predicted by Darcy's law and

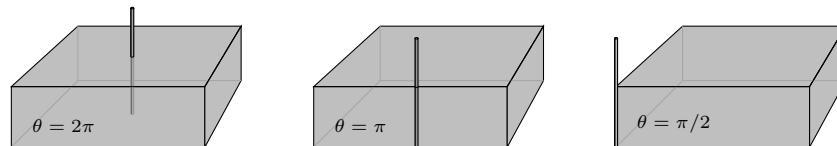


Fig. 11.27. The parameter θ measures the angular section of the wellbore that is connected to the reservoir and depends on where the well is placed inside a grid block.

actual drawdown resulting from formation changes in the near-well region. The typical range of skin factors is $-6 < S < 100$; negative values represent wells stimulated e.g., by hydraulic fracturing, whereas large positive values represent wells suffering from severe formation damage caused e.g., by various forms of clogging induced by drilling fluids, fines migration, precipitation of soluble salts, etc.

The mobility $\lambda_{\alpha,i}$ is computed differently for injection and production wells. For ***producing connections***, the mobility of a free phase of gas, oil, or water, is given as

$$\lambda_{\alpha,i} = \frac{(b_{\alpha}k_{r\alpha})_i}{\mu_{\alpha,i}}, \quad (11.20)$$

where the relative permeability $k_{r\alpha}$, the shrinkage/expansion factor b_{α} , and the phase viscosity μ_{α} are computed using pressure and saturation values from the grid block perforated by connection number i . If the produced gaseous or oleic phases contain vaporized oil or dissolved gas, respectively, the mobilities account for this as follows

$$\lambda_O = \lambda_o + R_v \lambda_g, \quad \lambda_G = \lambda_g + R_s \lambda_o. \quad (11.21)$$

For ***injecting connections***, MRST computes the mobility as the sum over all phases

$$\lambda_{\alpha,i} = y_{\alpha} \sum_{\alpha} \left(\frac{b_{\alpha}k_{r\alpha}}{\mu_{\alpha}} \right)_i. \quad (11.22)$$

where y_{α} is the volume fraction of phase α in the mixture *inside the wellbore*. This means that the injectivity is a function of the total mobility of the connected grid block.

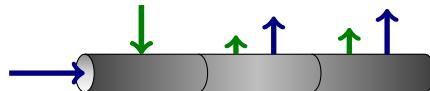


Fig. 11.28. Illustration of crossflow in a well; blue and green colors represent different fluid phases.

Wells connected to multiple grid blocks may sometimes experience ***cross-flow*** as illustrated in Figure 11.28. This means that fluids flow into some connections and out of other connections. Injectivities must therefore be modified so that they represent the mixture of the fluids inside the wellbore. To this end, we first compute inflow from producing connections at reference (stock-tank) conditions and use this to compute the *average* wellbore mixture at reference conditions. Then, we compute the average volumetric mixture at the conditions of each injection connection and use this to define the correct mobility. A similar average wellbore mixture is used to calculate the pressure head between the well's datum and each connection.

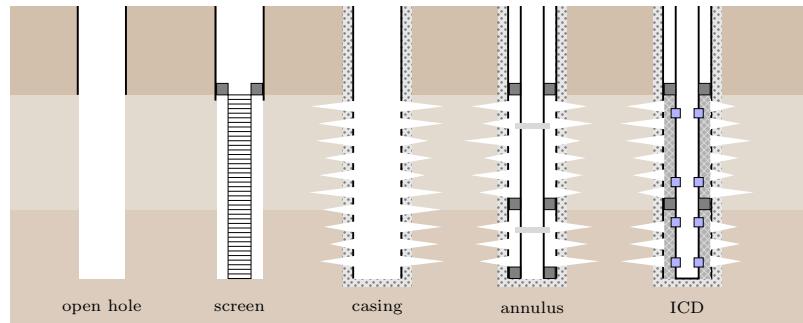


Fig. 11.29. Conceptual illustration of different types of well completions.

11.6.2 Multisegment wells

Many wells are designed so that the reservoir fluids do not flow directly from the sandface and into the wellbore. Oftentimes, the wellbore (production casing) lies inside another casing that has been cemented in place inside the drilled hole. The void *annular* space between the casing and the wellbore is limited by sealing packers that limit the annular space into sections and sleeves that provide flow paths between the annulus and the wellbore. One can also have various types of screens that aim to reduce/eliminate the inflow of solid particles that would cause erosion inside the wellbore. Simple versions of sand screens are made up of ribs, rods, or wire filters, possibly filled with gravel, whereas more modern ones consist of high-permeable, resin-coated gravel packed between two layers of wire-wrap filter media. In addition, the wells may have various components that can partially or fully choke the inflow to prevent unwanted fluids to enter the wellbore. Inflow control devices (ICDs) are passive or autonomous devices installed along the completion interval to optimize inflow along the length of the wellbore. These devices are designed to exert higher back pressure at high flow rates and can be used to equalize inflow from high and low-permeable zones, or delay water or gas breakthrough across selected intervals of horizontal wells. Inflow control valves serve the same purpose and are controlled from the surface through electric or hydraulic signals.

To further increase complexity, modern wells typically have directional profiles designed to intersect multiple targets with a single wellbore. Such wells can have multiple production zones, or consist of multiple deviated or horizontal branches (multilaterals). To increase the productivity of a well, it is common to install downhole pumps that lower the wellbore flowing pressure so that fluids can be drawn from the reservoir at a higher rate. If the downhole pressure is too low, it may be necessary to apply artificial gas lift, in which gas is injected downhole to lower the density of the flowing fluids and use gas expansion to help the produced fluids to the surface. Most oil wells require artificial lift at some point to ensure that they keep flowing when their bottom-

hole pressure decreases. Wells may also incorporate equipment for downhole compression and separation of fluid phases, which may be used to separate gas (or water) so that it can be reinjected downhole. Downhole processing can also be introduced to reduce the size and weight of surface or seafloor facilities or to overcome bottlenecks in existing surface processing systems.

All these different component will induce pressure drawdown and/or changes in the composition of the flowing fluids. To model this complex behavior, one may use a so-called *multisegment well model*, which can be thought of as a flow network consisting of a set of nodes connected by a set of segments (edges). Each node n has an associated volume and can store fluids, and fluids can flow along each segment connecting two nodes as described by a set of one-dimensional flow equations.

Continuous flow equations

The one-dimensional flow equations can either be formulated using phase velocities \vec{v}_α and volume fractions y_α , or component mixture velocities \vec{v}_c^m and mass fractions x_c for the gas, oil, and water components. Capillary pressure is negligible inside the wellbore and hence the conservation equations for volumetric quantities can be written as follows

$$\begin{aligned}\partial_t(b_w y_w) + \partial_x(b_w v_w) - b_w q_w &= 0, \\ \partial_t(b_o y_o + b_g r_v y_g) + \partial_x(b_o v_o + b_g r_v v_g) - b_o q_o - b_g r_v q_g &= 0, \\ \partial_t(b_g y_g + b_o r_s y_o) + \partial_x(b_g v_g + b_o r_s v_o) - b_g q_g - b_o r_s q_o &= 0.\end{aligned}\quad (11.23)$$

The phase velocities are not related to pressure gradients by Darcy's law, but rather through some nonlinear relation modelling friction, acceleration, inflow devices, valves, and so on. This relationship can be written

$$\nabla p - g\rho\nabla z - G(v, \rho, \mu) = 0, \quad (11.24)$$

where ρ , v , and μ are the mixture density, velocity, and viscosity, respectively, and G represents the (nonlinear) part of the pressure drop that cannot be explained by a Darcy relationship. The equivalent set of equations on mass form reads,

$$\partial_t(x_c \rho) + \nabla \cdot \vec{v}_c^m - q_c^m = 0, \quad c \in \{g, o, w\}. \quad (11.25)$$

Here, q_c^m is the mass source term of component c .

Discrete well equations

To define the discrete multisegment well models, we will utilize a vector notation analogous to the discrete operators we introduced in Section 4.4.2 for flow equations in the porous medium. Where we previously had two mappings F between cells and their bounding cell faces and C_1, C_2 between faces and

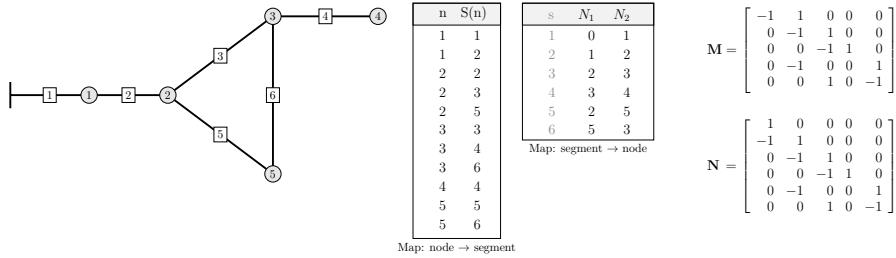


Fig. 11.30. Example of well topology and corresponding discrete operators.

adjacent cells, we now have similar mappings S from nodes to all connected segments and N_1, N_2 from segments to the adjacent nodes, see Figure 11.30. Given these mappings, the discrete gradient operator reads

$$\mathbf{grad}(\mathbf{v})[s] = \mathbf{p}[N_2(s)] - \mathbf{p}[N_1(s)]$$

and similarly the discrete divergence operator reads

$$\mathbf{div}(\mathbf{v})[n] = \sum_{s \in S(n)} \mathbf{v}[s] \mathbf{1}_{\{n=N_1(s)\}} - \sum_{s \in S(n)} \mathbf{v}[s] \mathbf{1}_{\{n=N_2(s)\}}.$$

These operators can both be represented as sparse matrices, almost like the D matrix. However, the top segment of a well since the top node corresponds to the reference depth at which the well's bottom-hole-pressure is defined. Hence, instead of calculating a pressure drop across the top segment, we impose a boundary condition on the top node corresponding to the correct control of the well. This means that the operators can be written $\mathbf{grad}(\mathbf{p}) = \mathbf{M}\mathbf{p}$ and $\mathbf{div}(\mathbf{v}) = -\mathbf{N}^T\mathbf{v}$, where \mathbf{M} equals \mathbf{N} except that the first row is removed. Discrete averaging operators are defined analogously. Given these operators, the discrete flow equations can be defined as follows: If we use the mass form, the conservation equation in each node reads

$$\frac{\mathbf{V}}{\Delta t} [\mathbf{x}_c \boldsymbol{\rho} - (\mathbf{x}_c \boldsymbol{\rho})^0] + \nabla \cdot (\mathbf{v}_c^m) - \mathbf{q}_c^m = \mathbf{0}, \quad (11.26)$$

where superscript 0 refers to values at the previous time step. The equation for the pressure drop reads,

$$\mathbf{grad}(\mathbf{p}) - g \mathbf{upw}(\boldsymbol{\rho}) \mathbf{grad}(\mathbf{z}) - G(\mathbf{v}^m, \mathbf{upw}(\boldsymbol{\rho}), \mathbf{upw}(\boldsymbol{\mu})) = \mathbf{0}. \quad (11.27)$$

Here, $\mathbf{grad}(\mathbf{z})$ is the gradient of the node depth, i.e., the depth difference between neighboring nodes. Note that (11.27) is not defined for the topmost segment and we need to specify a boundary or control equation at the top to close the coupled system (11.26)–(11.27).

Pressure-drop relations

The function G in (11.24) can be used to impose almost any pressure drop. For general network topologies, however, we assume that $\partial G / \partial v^m$ to be nonzero. As an example, the pressure drop over a *simple valve* can be modelled as

$$\Delta p = \frac{\rho u_c^2}{2C_v^2} \quad (11.28)$$

where u_c is the velocity in the constriction and C_v is the discharge coefficient. Another example is the *frictional pressure drop* along the wellbore, which may have a significant effect on the behavior of (long) horizontal wells. In ECLIPSE 100 [227], this pressure loss is given by

$$\Delta p = \frac{2fL}{D} \rho u^2, \quad (11.29)$$

where f is the Fanning friction factor, L is the length of the segment, D is the corresponding tubing diameter, ρ is the mixture density, and u is the mixture velocity. For a mixture mass flux v^m or mixture volume flux v , the velocity u is given by

$$u = \frac{v^m}{\rho \pi (D/2)^2} = \frac{v}{\pi (D/2)^2}. \quad (11.30)$$

For annular pressure drop, $(D/2)^2$ is replaced by $(D_o/2)^2 - (D_i/2)^2$, where D_o and D_i are the diameters of the outer and inner tubing, respectively. The Fanning friction factor depends on the Reynolds number $Re = \rho u D / \mu$ and is calculated as

$$\sqrt{f^{-1}} = -3.6 \log_{10} \left[\frac{6.9}{Re} + \left(\frac{e}{3.7D} \right)^{10/9} \right], \quad (11.31)$$

for $Re > 4000$ and $f = 16/Re$ in the laminar region for $Re < 2000$. In between, ECLIPSE interpolates between the corresponding end values for Re equal 2000 and 4000. MRST uses the same approach as ECLIPSE, but also allows use of (11.31) for all velocities. Coefficient e gives roughness of the tubing.

11.7 Black-oil simulation with MRST

Altogether, we have discussed a great number of details that must be brought together to make a simulator, even for fairly simple test cases. In addition, there are still a number of other important topics like preconditioning and details of boundary conditions, time chopping, nonlinear solution strategy, and so on, that I have *not* discussed in detail. As stated in the introduction of the chapter, I consider many of these details to be outside the scope of the book and I hence feel that time is right to switch the presentation to the AD-OO framework, which implements all the necessary ingredients entering industry-standard reservoir simulators. I therefore end the chapter by first

briefly outlining how the AD-OO framework can be used to simulate the SPE 1 benchmark model and then discussing some limitations and potential pitfalls that you, as a user of MRST and other reservoir simulators should be aware of. The next chapter will give a more systematic overview of the AD-OO framework and discuss more simulation examples.

11.7.1 Simulating the SPE 1 benchmark case

You have already encountered the fluid model of what is commonly referred to as the SPE 1 benchmark [204] multiple times in this chapter. The benchmark describes a depletion problem with gas injection in a small $10 \times 10 \times 3$ reservoir with a producer and an injector placed in diagonally opposite corners. The porosity is uniform and equal 0.3, whereas the permeability is isotropic with values 500, 50, and 200 md in the three layers with thickness 20, 30, and 50 ft. The reservoir is initially undersaturated with a pressure field that is constant in each layer, a uniform mixture of water ($S_w=0.12$) and oil ($S_o=0.88$) with no initial free gas ($S_g=0.0$) and a constant solution gas-oil ratio ($R_s \approx 226.2$) throughout the model. Although the problem is really a two-phase gas/oil problem describing gas injection into an undersaturated oil, the input file describes a three-phase problem with almost stationary water. The original problem was posed to study ten years of production, but herein we only report and compare solutions for the first 1216 days.

The steps necessary to read and interpret the ECLIPSE input data and create the necessary objects representing grid, petrophysical and fluid properties have been discussed in detail earlier in the book:

```
gravity reset on
deck = readEclipseDeck(fullfile(getDatasetPath('SPE1'), 'BENCH_SPE1.DATA'));
deck = convertDeckUnits(deck);
G    = computeGeometry(initEclipseGrid(deck));
rock = compressRock(initEclipseRock(deck), G.cells.indexMap);
fluid = initDeckADIFluid(deck);
```

In addition, we need to set up an initial state. The **SOLUTION** section in the input file specifies how the reservoir should be initialized. Here, however, hard-code the fluid equilibrium state:

```
[k, k] = ind2sub([prod(G.cartDims(1:2)), G.cartDims(3)], G.cells.indexMap);
p0      = [329.7832774859256 ; ... % Top layer
          330.2313357125603 ; ... % Middle layer
          330.9483500720813 ]; % Bottom layer
p0      = convertFrom(p0(k), barsa);
s0      = repmat([ 0.12, 0.88, 0.0 ], [G.cells.num, 1]);
rs0    = repmat( 226.1966570852417 , [G.cells.num, 1]);
rv0    = 0; % dry gas

state = struct('s', s0, 'rs', rs0, 'rv', rv0, 'pressure', p0);
```

The first statement creates a partition vector containing the layer index, which we use to lookup in the vector of equilibrium pressures values. All the statements above have been collected in a utility function `setupSPE1` in the `ad-blackoil` module.

The next step is to create a model object

```
model = selectModelFromDeck(G, rock, fluid, deck);
```

Technically, the model objects in AD-OO are defined using MATLAB's `classdef` and includes a number of properties and a number of methods that can operate on input parameters and the data contained within the object. Classes can also define events and enumerations, but these are not used in AD-OO. Those not familiar with the concept of objects, can think of `model` as a standard `struct` that has both data members and function pointers. In this particular case, the input data will lead to the creation of a three-phase, black-oil model with dissolved gas but no vaporization of oil. The following is a list of a few of the data members

```
ThreePhaseBlackOilModel with properties:
    disgas: 1
    vapoil: 0
    fluid: [1x1 struct]
    rock: [1x1 struct]
    water: 1
    gas: 1
    oil: 1
    saturationVarNames: {'sw' 'so' 'sg'}
    gravity: [0 0 9.8066]
    FacilityModel: [1x1 FacilityModel]
    operators: [1x1 struct]
    G: [1x1 struct]
```

Here, we recognize `G`, `rock`, and `fluid`, which are the standard MRST objects we constructed to represent grid, petrophysics and fluid properties. The `operators` structure contains transmissibilities, pore volume, and discrete operators and is the same structure we referred to as `s` in the previous section. `FacilityModel` is another object that will be used to describe well models (and surface facilities). Last, but not least, the object contains five flags telling which of the three phases are present and whether gas is allowed to dissolve in oil and oil is allowed to vaporize into gas. The last two flags were used in the codes we presented in the previous section for evaluating PVT properties of the oleic and gaseous phases. This code consists of excerpts from the function `equationsBlackOil` and a number of helper functions that has been simplified and reorganized for pedagogical purposes. The AD-OO framework never calls this function directly, but instead accesses it indirectly through a generic member function called `getEquations`. This way, the parts of the framework that implements the time loop and the nonlinear solver does not need to know any specifics of the model as long as we know that the model exists and that

calling `getEquations` will compute the corresponding flow equations on residual form with automatic differentiation activated for all primary unknowns. In addition to the parameters outlined above, the model contains a number of other parameters that control the nonlinear solver, chopping of time steps, and output of results.

The next thing we have to do is to create a simulation *schedule*, which essentially consists of a set of time steps and a specification of the drive mechanisms that will be present during each of these time steps. These are described in the **SCHEDULE** section of the input deck, which for SPE 1 declares two different wells:

```
-- WELL SPECIFICATION DATA
-- WELL GROUP LOCATION BHP PI
-- NAME NAME I J DEPTH DEFN
WELSPCS
  'P' 'G' 10 10 8400 'OIL' /
  'I' 'G' 1 1 8335 'GAS' /
/
-- COMPLETION SPECIFICATION DATA
-- WELL -LOCATION- OPEN/ SAT CONN WELL
-- NAME I J K1 K2 SHUT TAB FACT DIAM
COMPDAT
  'P' 10 10 3 3 'OPEN' 0 -1 0.5 /
  'I' 1 1 1 1 'OPEN' 1 -1 0.5 /
/
```

The **WELSPCS** keyword shown to the left introduces the wells, defines their name, the position of the well head, the reference depth for the bottom holes, and the preferred phase. The keyword **COMPDAT** shown to the right, specifies how each well is completed, i.e., in which cells each well can inject/produce fluids. The next we need to define is how the wells are controlled during the simulation:

```
-- PRODUCTION WELL CONTROLS
-- WELL OPEN/ CNTL OIL WATER GAS LIQU RES BHP
-- NAME SHUT MODE RATE RATE RATE RATE RATE
WCONPROD
  'P' 'OPEN' 'ORAT' 20000 4* 1000 /
/
-- INJECTION WELL CONTROLS
-- WELL INJ OPEN/ CNTL FLOW
-- NAME TYPE SHUT MODE RATE
WCONINJE
  'I' 'GAS' 'OPEN' 'RATE' 100000 100000 50000 /
/
```

Here, specification says that the producer should be controlled by oil rate, whereas the injector should inject gas and be controlled by rate.

To turn this input into a data object describing the simulation schedule in a form that MRST understands, we issue the following command:

```
schedule = convertDeckScheduleToMRST(model, deck);
```

which parses the `deck` structure and creates the following data structure:

```
schedule =
  step: [1x1 struct]      schedule.step =
  control: [120x1 double]    control: [120x1 double]
  control: [1x1 struct]      val: [120x1 double]      schedule.control =
                                W: [2x1 struct]
                                bc: []
                                src: []
```

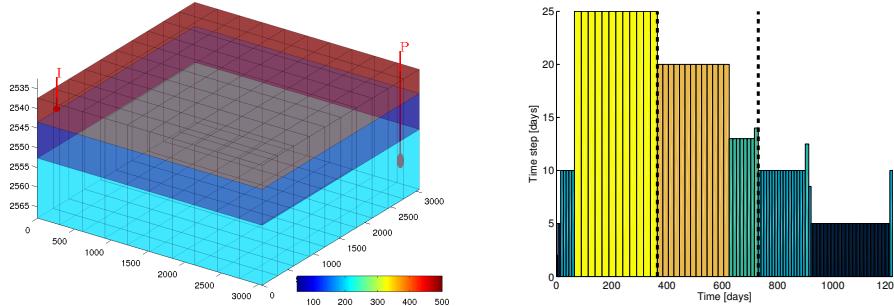


Fig. 11.31. Horizontal permeability and well positions for the SPE 1 benchmark case. The injector is perforated in the upper and the producer in the lower layer. The right plot shows the time steps in the simulation schedule.

The schedule has two data members: a list of steps and a list of controls. The controls specify a set of *targeted*² drive mechanisms (wells, boundary conditions, and source terms) that each is unique and will be applied during at least one step. Each step consists of a time-step length (`val`) and the number (`control`) of the corresponding control. In our case, there is only one control consisting of the injection and production wells described above (see Figure 11.31), which will be active during each of the 120 time steps. The time steps are given by the keyword `TSTEP`, which can be repeated to specify the full schedule (see Figure 11.31):

<pre> TSTEP 1.0 2*2.0 2*5.0 5*10.0 11*25.0 / TSTEP 25.0 / </pre>	<pre> -- INITIAL STEP: 1 DAY, MAXIMUM: 6 MONTHS TUNING 1 182.5 / 1.0 0.5 1.0E-6 / / </pre>
--	--

The `TUNING` keyword gives parameters for time-step control, time truncation and convergence control (here: mass-balance error $\leq 10^{-6}$), and control of Newton and linear iterations (not given here).

To run the schedule, we must select a nonlinear solver, which is implemented as a class in the AD-OO framework:

```
nls = NonLinearSolver('useLineSearch', true);
```

The `NonLinearSolver` class is a general nonlinear solver based on Newton's method. It is capable of time-step selection and chopping based on convergence rates and can be extended via subclassing or modular linear solvers and time-step classes. Convergence is handled by the model. The `NonLinearSolver` simply responds based on what the model reports in terms of convergence to ensure some level of encapsulation. Here, we initialize the solver class to use

² This is an important distinction: the schedule may for instance request that a given well is operated according to a prescribed rate (rate control). However, if this rate cannot be achieved without violating physical limits on the bottom-hole pressure, the simulation will switch to using a bottom-hole control instead.

line-search, as discussed in Section 10.2.2. The following excerpts from the `nls` object

```
NonLinearSolver with properties:
    maxIterations: 25
    maxTimestepCuts: 6
        LinearSolver: [1x1 BackslashSolverAD]
    timeStepSelector: [1x1 SimpleTimeStepSelector]
        useLinesearch: 1
            verbose: 0
            :
:
```

show that the nonlinear solver will at most use 25 iterations, be allowed to cut the time step six times before giving up, use the `BackslashSolverAD` linear solver and the `SimpleTimeStepSelector` to select time steps, use line-search as specified, run in non-verbose mode so that no details about convergence is reported to screen, and so on.

We are now ready to run the full simulation and specify that we output of well solutions, reservoir states, and a report that contains details about time steps, convergence, etc:

```
[wellSols, states, report] = ...
    simulateScheduleAD(state, model, schedule, 'nonlinearsolver', nls);

Solving timestep 001/120:           -> 1 Day
Solving timestep 002/120: 1 Day      -> 3 Days
:
:
Solving timestep 120/120: 3 Years, 111 Days -> 3 Years, 121 Days
*** Simulation complete. Solved 120 control steps in 103 Seconds .. ***
```

To inspect the solutions, we issue the following commands, which will bring up the two graphical user-interfaces shown in Figure 11.32:

```
plotWellSols(wellSols, report.ReservoirTime)

figure;
plotToolbar(G, states)
plotWell(G, schedule.control(1).W, 'radius', .5)
axis tight, view(-10, 60)
```

Let us study the dynamics of the reservoir in some detail. Figure 11.33 shows how the bottom-hole pressure in the injector and the solution gas-oil ratio inside the reservoir evolve as functions of time. Before production starts, the reservoir is in an undersaturated state, shown as a green marker in the plots of `rs` as function of `p0`. Initially, the injector needs a relatively high bottom-hole pressure (bhp) to force the prescribed gas rate into the reservoir. Because the reservoir is undersaturated, the injected gas will dissolve in the oil phase, so that the viscosity decreases. A lower bhp is thus required to maintain the injection rate. As the injection continues, cells near the well will gradually become saturated so that free gas is liberated and starts accumulating at the

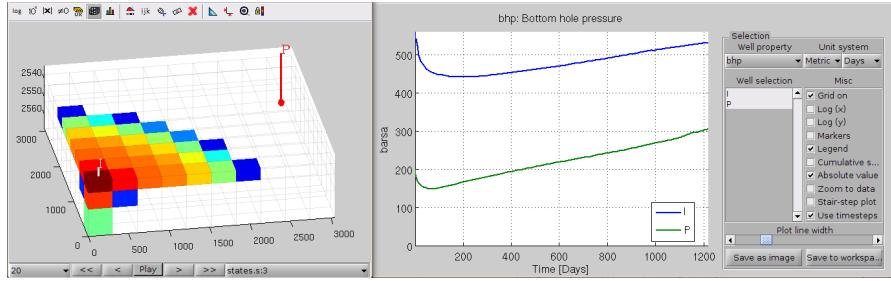


Fig. 11.32. Graphical user interfaces for plotting reservoir data and well responses. The left plot shows the saturation of free gas after 20 time steps. The right plot shows the variation in bottom-hole pressure in the injector and producer throughout the simulation.

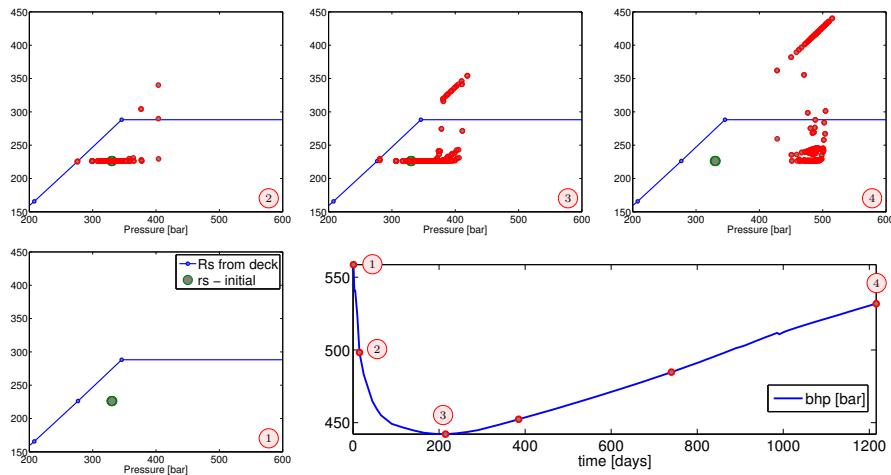


Fig. 11.33. Reservoir dynamics of the SPE 1 case during the first three years. The lower-right plot shows the bottom-hole pressure of the injector as function of time. The other plots show the solution gas-oil ratio as function of pressure for all cells in the reservoir at four different instances in time.

top of the reservoir. Similarly, some free gas will be liberated near the producer because of the pressure drawdown. As more gas is injected into the reservoir, more cells transition from an undersaturated to a saturated state, giving an increasing plume of free gas that expands out from the injector along the top of the reservoir (see the left plot in Figure 11.32). The total injection rate is higher than the production rate, and hence the pressure in the reservoir will increase, and after approximately 200 days, we see that the injector bhp starts to increase again to maintain the specified rate.

11.7.2 Comparison against a commercial simulator

To build confidence in our simulator, we can verify the simulation above against ECLIPSE. To compare the two, we first extract surface rates and bottom-hole pressures from MRST as plain arrays:

```
[qWs, qOs, qGs, bhp] = wellSolToVector(wellSols);
```

We also extract the time steps and determine the indices of the injector and producer:

```
T = convertTo(cumsum(schedule.step.val), year);
inj = find([wellSols{1}.sign] == 1);
prod = find([wellSols{1}.sign] == -1);
```

Output from ECLIPSE is usually stored in so-called summary files, which can be used to reinitialize a simulation so that it can be continued. The `deckformat` module in MRST offers useful functionality for reading and interpreting such (binary) output files (see `readEclipseSummaryFmt` and `readEclipseSummaryUnFmt`). To simplify the comparison for the SPE 1 benchmark, the data set supplied with MRST includes a `*.mat` file with selected well responses obtained by simulating the exact same input deck with ECLIPSE 100. These data are stored as a MATLAB structure, which also contains a few functions for extracting the raw data. We load the summary

```
load SPE1_smry
ind = 2:size(smry.data,2);
Te = smry.get(':+:+:+:', 'YEARS', ind);
```

Here, we have ignored the first entry, which contains zero values. Figure 11.34 shows a comparison of producer gas/oil ratio and bottom-hole pressure in the producer. These are extracted as follows:

```
eGOR = convertFrom(smry.get('PRODUCER', 'WGOR', ind), 1000*ft^3/stb)';
eBHPp = convertFrom(smry.get('PRODUCER', 'WBHP', ind), psia)';
eBHPi = convertFrom(smry.get('INJECTOR', 'WBHP', ind), psia);

mGOR = qGs(:,prod)./qOs(:,prod);
mBHPp = bhp(:,prod);
mBHPi = bhp(:,inj);
```

As we see from the figure, there is (almost) perfect match between MRST and ECLIPSE, which is what we should expect, given that the two simulators implement (almost) exactly the same numerical methods.

11.7.3 Limitations and potential pitfalls

Numerical flow models based on the black-oil equations and their like are complex and delicate creatures that should be treated cautiously to ensure

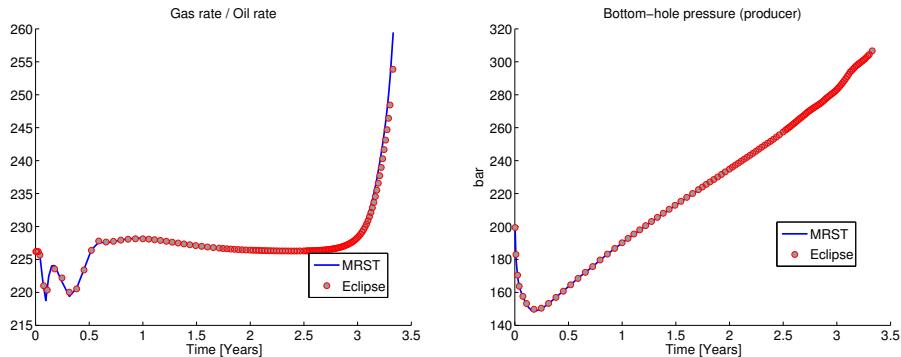


Fig. 11.34. Comparison of well responses for the SPE 1 benchmark simulated by MRST and ECLIPSE 100.

that simulations make sense. A commercial simulator like ECLIPSE has a large number of precautions built into it, so that the simulator on one side will refuse to simulate unphysical situations, but on the other side, manage to give an answer even though the user gives incomplete or inconsistent data. (You may also recall our discussion of time-step chops implemented to ensure convergence when faced with severe nonlinearity, which potentially *could* be the result of an inadequate model or incorrect parameters.) Being aware of limitations and potential pitfalls is important, in particular when using a prototyping tool like MRST that has very few precautions built into it, but also when using mature commercial tools, whose alluring robustness may sometimes be deceptive. In this section, I will try to review a few of the potential issues and comment on possible precautions.

When can black-oil models be applied?

Any model is no better than its assumptions, and it is important to know what the limitations of a model is so that it is not used for the wrong purposes. Black-oil models can, in principle, be applied to simulate all the five types of reservoirs outlined in Section 11.3.2, as long as these are produced by depletion or waterflooding. As a general rule of thumb, one should be more precautions when applying black-oil models to gas-injection processes that involve significant changes in the composition of the oil and gas pseudo-components during the displacement. Vaporization and retrograde condensation in rich gases are examples of such processes, and here one should be careful to check the validity of black-oil models against more comprehensive compositional models. To ensure that the black-oil description is sufficiently accurate, the recovery process should satisfy the following characteristics [228]:

- The recovery process should be isothermal.
- The path taken by the reservoir fluids should be far from the critical point.

- The amount of liberated gas or condensate dropout should be a small part of the hydrocarbon in place.
- The remaining hydrocarbon composition should not change significantly when gas is liberated or vaporized condensate drops out.

If these conditions are not satisfied, one should use a compositional simulator.

Consistent PVT parameters

The PVT parameters used in black-oil models are originally derived from laboratory analysis of (recombined) reservoir fluids, but in recent years it has become more common to derive the necessary pressure-dependent parameters by simulating differential liberation experiments by use of compositional models with appropriate equations of state. Regardless of their source, the derived PVT models must satisfy certain physical properties, see [63]. First of all, the total compressibility of the system must be non-negative, i.e.,

$$c_t = S_w c_w + S_o c_o + S_g c_g + c_r \geq 0,$$

where c_r is the rock compressibility and c_α is the isothermal compressibility of phase α ,

$$\begin{aligned} c_w &= -\frac{B'_w}{B_w}, & c_o &= -\frac{B'_o}{B_o} + \frac{(B_g - B_o R_v) R'_s}{B_o (1 - R_s R_v)}, \\ c_g &= -\frac{B'_g}{B_g} + \frac{(B_o - B_g R_s) R'_v}{B_g (1 - R_s R_v)}. \end{aligned}$$

Since this property has to be valid for all saturations, it follows that each of the isothermal compressibilities also must be nonnegative, which gives the following three consistency conditions

$$\begin{aligned} B'_w &\leq 0, \\ (1 - R_s R_v) B'_o &\leq (B_g - R_v B_o) R'_s, \\ (1 - R_s R_v) B'_g &\leq (B_o - R_s B_b) R'_v. \end{aligned} \tag{11.32}$$

To make sure that the partial volumes in (11.11) are also positive, the following additional conditions must be satisfied

$$R_s R_v < 1, \quad B_w > 0, \quad B_g - R_v B_o > 0, \quad B_o - R_s B_g > 0. \tag{11.33}$$

As we have seen in the discussion above, B_o , R_s , and R_v are normally increasing functions with positive derivatives, whereas B_g and B_w are decaying functions with negative derivatives.

Use of tabulated data

Use of tabulated data, as discussed above, is very common in industry-grade simulation models. This means that when the simulator requires values between data points, these need to be interpolated. Likewise, if values are requested outside the range of tabulated data, the data need to be extrapolated. Algorithmic choices of how data are interpolated and extrapolated are specific to the individual simulator, but can have a significant impact on both the accuracy and stability of the simulator, and the user should therefore be aware of what methods are used.

The default choice in ECLIPSE is to use linear interpolation and zeroth order extrapolation for rock-fluid properties, so that the simulator picks the value of the first or last tabulated point rather than using trend interpolation based on the closest points. This means that *derivatives* of tabulated quantities are piecewise constants and effectively are computed using finite-differences between tabulated points. PVT properties are also interpolated within their range of definition, but properties like R_s and R_v may also be extrapolated. In MRST, we have chosen to follow the interpolation conventions from ECLIPSE in the `ad-props` module to be able to produce simulation results that are as identical as possible when using the same discretizations and solvers. This raises several issues you should be aware of:

- First of all, you should make sure that the data points cover the value range of the simulation. For PVT properties, this usually means that data should be defined for higher pressures than what are initially found in the reservoir, since pressure can increase during the simulation as a result of fluid injection, gravity segregation, etc. In ECLIPSE 100, you can use the keyword `EXTRAPMS` to force the simulator to issue a warning whenever it extrapolates outside of tabulated data. Such a safeguard is not yet implemented in MRST.
- To reduce numerical errors from the numerical interpolation, make sure that data points are sufficiently dense and distributed so that interpolation errors do not vary too much in magnitude throughout the parameter range.
- Try to avoid discontinuities, strong kinks, and other abrupt changes in tabulated properties unless these are dictated by physics (see, e.g., the relative permeabilities for SPE 9 in Figures 11.6 and 11.7). If necessary, introduce extra data points near strong nonlinearities and local extremal points to avoid introducing unphysical oscillations and too large values for derivatives. This will improve the convergence of the nonlinear solver and reduce the number of (unnecessary) time-step chops.
- On the other hand, introducing too many points will make table lookup more expensive and contribute to slow down your simulation. For MRST, you should be particularly beware of multiple regions. Property evaluation is vectorized over all cells to make it as efficient as possible, but with multiple relperm or PVT regions, each region must be processed sequentially

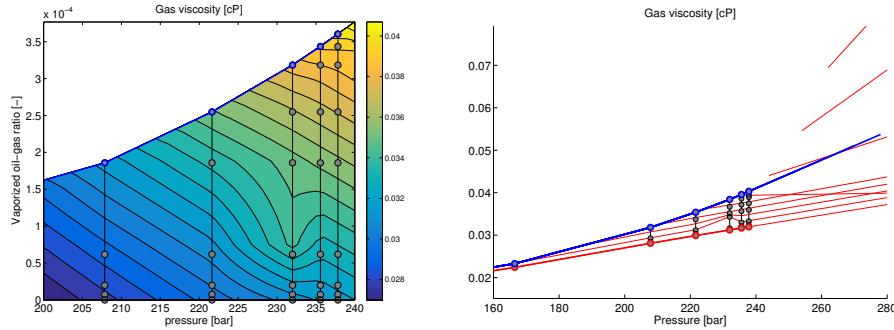


Fig. 11.35. Interpolated and extrapolated gas viscosity for the SPE 3 fluid model. The left plot shows contours of μ_g over parts of the (p, R_v) domain, with data points shown as filled circles. The right plot shows μ_g as function of pressure, with red lines corresponding to constant R_v values.

in a **for** loop, and this can add a significant computational overhead if the number of regions is large³.

- In certain cases, ECLIPSE interpolates functions of the input data or functional *combinations* of multiple properties, rather than the tabulated values. The primary example is the PVT data for oil. Here, ECLIPSE 100 interpolates reciprocals of B_o and $\mu_o B_o$, whereas ECLIPSE 300 interpolates reciprocals of B_o , but interpolates μ_o directly. Using different interpolations may have a significant effect on simulation results.

To give you an idea of how things can go wrong, let us revisit two of the SPE benchmark models, starting with SPE 3. To sample the model, we make a 101×21 regular mesh of (p, R_v) values covering the rectangular domain $[p_0, p_1] \times [0, R_v(p_1)]$ for $p_0 = 80$ and $p_1 = 280$ bar. In the input data, properties are only defined for pressures in the interval $[83.75, 237.82]$ bar, and hence we need to use both interpolation and extrapolation to compute values. Figure 11.35 reports the computed gas viscosities for values toward the upper end of the pressure interval. Here, the left plot clearly shows that the viscosity is not necessarily a monotone function of pressure for fixed R_v . More important, the right plot shows that we get incorrect behavior with highly unrealistic viscosities for R_v values less than $R_v(p)$ once the pressure exceeds maximum data values by more than a few percent.

Figure 11.36 shows that the SPE 1 model will violate the consistency condition $B_o - R_s B_g > 0$ in (11.33) for high pressure values, as observed previously by Coats [63]. By convention, the formation-volume factor B_o of oil is computed by interpolating the shrinkage factor b_o rather than the tabulated B_o values. In the tabulated data, b_o declines with increasing R_s values and

³ One notorious example is the Norne simulation model, which prescribes a redundant end-point scaling of the relative permeabilities in each and every grid cell, which makes simulation in MRST very costly.

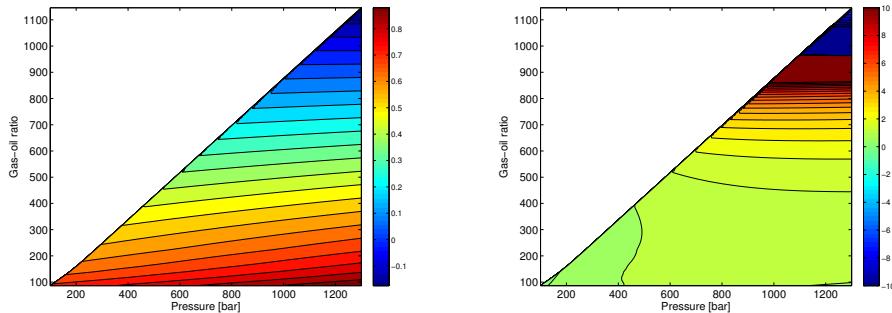


Fig. 11.36. Inconsistency in the PVT data of the SPE 1 benchmark giving negative partial volumes. The left plot shows the interpolated shrinkage factor b_o , whereas the right plot shows the resulting values of $B_o - R_s B_g$. To emphasize the blowup and the negative values, the color scale has been cropped to $[-10, 10]$, but the actual values are up to two orders larger.

will hence cross zero for sufficiently high values. This leads to blowup and negative values for B_o . This behavior is only observed for (p, R_s) values that are larger than what is to be expected in practical computations, but it is still interesting to note this artifact.

Initialization and equilibration

Setting the correct initial state is obviously essential to obtain accurate predictions of how a reservoir will behave under production. ECLIPSE input offers two ways of setting up the initial state:

- you can either input individual cell values of the initial state using the keywords PRESSURE, SWAT, SGAS, and RS or PBUB in an ECLIPSE input file, or
- you can calculate the initial state equilibration, i.e., determine it on the basis of hydrostatic equilibrium.

At the time of writing, MRST reads individual cell values, but only offers limited functionality for equilibration calculations. In the previous chapter we saw how variations in rock properties will cause variations in capillary pressures. The observed fluid contacts (gas-oil and oil-water) will generally show vertical variation and not always be planar. To determine the free fluid contacts, we need to know the phase pressures inside the reservoir. The level of free gas is then defined as the depth at which the gas and oil pressures coincide, i.e., the depth at which the oil-gas capillary pressure is zero. The level of free water is defined analogously.

Typical input data to an equilibration calculation consist of: pressure at a datum depth, gas-oil and oil-water contacts, capillary pressure at each contact, and gas-oil ratio (R_s) or bubble-point pressure (p_b) as function of depth. Given

these data, it is, in principle, a straightforward task to determine the fluid distribution, which generally will consist of five different zones:

1. A gas cap, in which gas is the only continuous phase ($S_g = 1 - S_{wi}$)

$$\frac{dp_g}{dz} = g\rho_g, \quad p_o = p_g - P_{cog}(S_g), \quad p_w = p_o - P_{cow}(S_{wi})$$

2. An oil/gas transition zone, in which both gas and oil are continuous:

$$\frac{dp_g}{dz} = g\rho_g, \quad \frac{dp_o}{dz} = g\rho_o, \quad p_w = p_o - P_{cow}(S_{wi})$$

The saturations are $S_w = S_{wi}$, $S_g = P_{cog}^{-1}(p_g - p_o)$, and $S_o = 1 - S_g - S_{wi}$.

3. An oil zone, in which oil is the only continuous phase ($S_o = 1 - S_{wi}$):

$$\frac{dp_o}{dz} = g\rho_o, \quad p_g = p_o + P_{cog}(0), \quad p_w = p_o - P_{cow}(S_{wi})$$

4. An oil/water transition zone, in which both oil and water are continuous:

$$\frac{dp_o}{dz} = g\rho_o, \quad \frac{dp_w}{dz} = g\rho_w, \quad p_g = p_o + P_{cog}(0)$$

The saturations are $S_g = 0$, $S_w = P_{cow}^{-1}(p_o - p_w)$, and $S_o = 1 - S_w$.

5. A water zone, in which only water is present:

$$\frac{dp_w}{dz} = g\rho_w, \quad p_o = p_w + P_{cow}(1), \quad p_g = p_o - P_{cog}(0)$$

In practice, however, the equilibration procedure is complicated by the fact that the fluid contacts may vary between different compartments in the reservoir and that capillary pressure curves may vary from one cell to the next. Notice also that numerical artifacts and inaccuracies may arise if the grid is not aligned with the fluid contacts, see e.g., [214].

Once your model is initialized, it is a good precaution to run a simulation with no external drive mechanisms applied to see if the reservoir contains any movable fluids, i.e., check whether the initial phase and pressure distributions remain invariant in time.

12

The AD-OO framework

Earlier in the book we have shown that combining a fully implicit formulation with automatic differentiation ensures that it is simple to extend basic flow models with new constitutive relationships, extra conservation equations, new functional dependencies, and so on. By using numerical routines and vectorization from MATLAB combined with discrete differential and averaging operators from MRST, these equations can be implemented in a very compact form close to the mathematical formulation. Writing a simulator as a single script, like we did for single-phase flow in Chapter 7 or for two-phase flow without phase transfer in Section 11.1, has the advantage that the code is self-contained, quick to implement, and easy to modify as long as you work with relatively simple flow physics. However, as the complexity of the flow models and numerical methods increases, and more bells and whistles are added to the simulator, the underlying code will inevitably become cluttered and unwieldy.

The AD-OO framework was introduced to structure our fully implicit simulators and enable rapid prototyping also for more complex flow physics. The previous chapter introduced you briefly to this framework and showed a simple example of how you can use it to quickly set up a simulation from an ECLIPSE input file. Altogether, the AD-OO framework offers a lot of the features that can be found in commercial simulators, and understanding its design and all the nitty gritty details of the actual implementation will obviously provide you with a lot of valuable insight into practical reservoir simulation. This chapter starts with a somewhat detailed discussion of the most central classes and member functions to demonstrate the general philosophy behind the framework, but does not discuss in detail how boundary conditions, source terms, and wells are implemented. We then end the chapter with a few simulation examples to outline the functionality and technical details you need to set up your own black-oil simulations. In particular, you will see how to set up simulation models both with and without the use of ECLIPSE input. Hopefully, the presentation will also provide you with sufficient detail to start developing new simulators with other types of flow physics.

12.1 Overview of the simulator framework

Looking back at the procedural implementations presented so far in the book, there are a number of data objects that have been introduced to keep track of all the different entities making up a simulation model:

- a state object holding the unknown pressures, saturations, concentrations, inter-cell fluxes, and unknowns associated with the wells;
- the grid structure `G` giving the geometry and topology of the grid;
- a structure `rock` representing the petrophysical data: primarily porosity and permeability, but also net-to-gross and multipliers that limit the flow between neighboring cells;
- a structure `fluid` representing the fluid model, having a collection of function handles that can be queried to give fluid densities and viscosities, evaluate relative permeabilities, formation volume factors, etc;
- additional structures that contain the global drive mechanisms: wells, volumetric source terms, boundary conditions;
- and optionally a structure that contains discrete differentiation and averaging operators.

In a procedural code, these data objects will be passed as arguments to solvers or functions performing sub-tasks like computing residual equations. As an example, all incompressible flow solvers in the `incomp` family require the same set of input parameters: reservoir state, grid and transmissibilities, and a fluid object plus optional parameters describing the drive mechanisms. The transport solvers, on the other hand, require state, grid, fluid, and rock properties. Implicit solvers for the fully equation system require the union of these quantities to set up the residual equations. An obvious simplification would be to collect all data describing the reservoir and its fluid behavior in a `model` object, which also implements utility functions to access model behavior. Given a reservoir state, for instance, we can then query values for physical variables with a syntax of the form

```
[p0, sW, sG, rs, rv] = ...
    model.getProps(state, 'pressure', 'water', 'gas', 'rs', 'rv');
```

As we saw in the previous chapter, which among s_g , r_s , and r_v we should choose as primary reservoir variable will vary from cell to call depending on the fluid phases present. Ideally, the model object should keep track of this and present us with the correct vector of unknowns for all cells. It is also natural that the model object should be able to compute the flow equations in residual form, e.g., by a call on the form

```
eqs = model.getEquations(oldState, newState, dT);
```

By introducing a generic interface like this, we can develop nonlinear solvers that require no specific or very limited knowledge of the physical model. A next step would be to separate the time-stepping strategy and the linear solver from

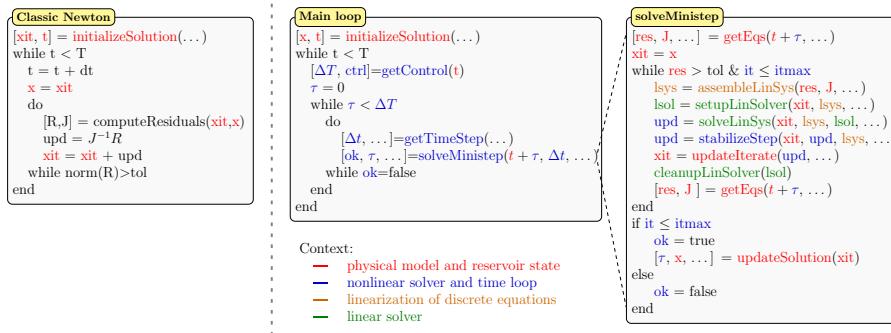


Fig. 12.1. Time loop of a simulator for a simple Newton solver (left) and for a more sophisticated solver with stabilization and time-step control (right). For the latter, the iteration has been divided into various numerical contexts.

the nonlinear solvers, so that we easily can switch between different methods or replace a generic method by a tailor-made method. A critical evaluation of our implementations also shows that large fractions simulator scripts are devoted to plotting of results. Likewise, implementing variations of the same model will either result in code duplication or a large number of conditionals that tend to clutter the code.

A standard approach to overcome these difficulties is to use object-orientation. The object-oriented, automatic-differentiation (AD-OO) framework in MRST has been tailor-made to support rapid prototyping of new reservoir simulators based on fully or sequentially implicit formulations. An essential idea of the framework is to develop general model and simulator classes and use inheritance to develop these into simulators for specific equations. The framework splits

- physical models describing the porous medium and the fluids flowing in it,
- nonlinear solvers and time-stepping methods,
- linearization of discrete equations, which have been formulated using generic discretization, averaging, and interpolation operators as discussed previously,
- solution of the linear system

into different numerical context. This way, we only expose needed details and enable more reuse of functionality that has already been developed.

Figure 12.1 illustrates how these contexts may appear in the main loop of a simulator. The left part of the figure shows an unsophisticated Newton loop of the type used for single-phase flow in Chapter 7, which consists of an outer loop running the time steps and an inner loop performing the Newton iteration for each time step. The right part shows a more advanced nonlinear solver which utilizes some kind of time-control mechanism to run the targeted control steps of a simulation schedule. In this setup, the *physical model* is re-

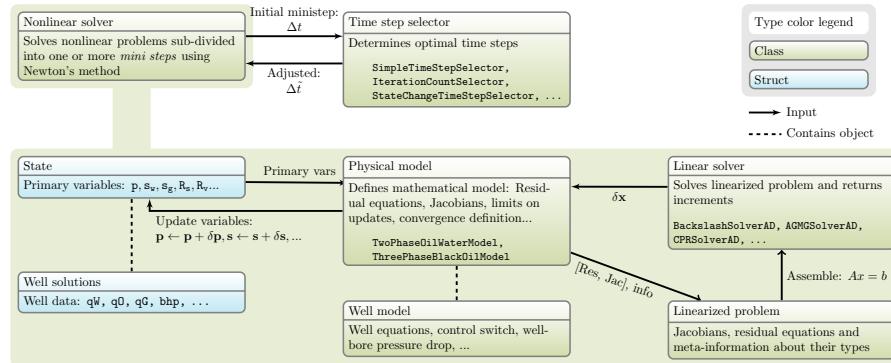


Fig. 12.2. Data diagram for a single nonlinear solve with time-step estimation in the AD-OO framework.

sponsible for initializing the reservoir state. The *nonlinear solver* implements the mechanism that selects and adjusts the time steps and performs a nonlinear iteration for this time step. Inside this nonlinear iteration, the *physical model* computes residual equations defined over all cells and possibly also cell faces and discrete entities in the well representation (connections or nodes and segments). When the residual equations are evaluated, the corresponding Jacobian matrix containing the linearized residual equations is computed implicitly by the AD library. This produces a collection of matrix blocks that each represents the derivative of a specific residual equation with respect to a primary variable. The *linearization* context assembles the matrix blocks into an overall Jacobi matrix for the whole system, and if needed eliminates certain variables to produce a reduced linear system. The *linear solver* sets up appropriate preconditioners and solves the linear system for the Newton increments, while the *nonlinear solver* performs necessary stabilization, e.g., in the form of an inexact line-search algorithm. Finally, the *physical model* updates reservoir states with computed increments and recomputes residuals to be checked against prescribed tolerances by the *nonlinear solver*.

Figure 12.2 shows how a single nonlinear solve with time-step estimation is realized in terms of classes and structures in the AD-OO framework. The *ad-core* module utilizes functionality from *mrst-core* to implement a generic nonlinear solver, a set of generic time-step selectors, and a set of linear solvers. The nonlinear solver does not know any specific details of the physical model, except that is able to perform the generic operations outlined above; we will come back to model classes in the next section. The time-step selectors implement simple heuristic algorithms like the Appleyard and modified Appleyard chop as used in commercial simulators. The linear solver class provides an interface for various solvers including MATLAB's default backslash solver as well as a state-of-the-art, constrained pressure residual (CPR) preconditioner [253, 95], which can be combined with efficient algebraic multigrid solvers like

the aggregation-based method of [203]. Assembly of the linearized system is relegated to a special class that stores meta-information about the primary variables and the residual equations (i.e., whether they are reservoir, well, or control equations). This information is can be used by the linear solver class to help setting up preconditioning strategies that exploit special structures in the problem.

Example 12.1 (1D Buckley–Leverett problem). To illustrate the use of the non-linear solver, let us revisit the classical Buckley–Leverett problem from Example 9.3 on page 292 and Section 10.3.1. For completeness, we show all code lines necessary to set up this problem in the AD-OO framework. First, we construct the reservoir and fluid data:

```
mrstModule add ad-core ad-props ad-blackoil
G      = computeGeometry(cartGrid([50, 1, 1], [1000, 10, 10]*meter));
rock  = makeRock(1*darcy*ones, .3);
fluid = initSimpleADIFluid('phases', 'W0', 'n', [2 2]);
```

We then create a two-phase black-oil model with constant shrinkage factors (from `ad-blackoil`) and set up the initial state:

```
model = TwoPhaseOilWaterModel(G, rock, fluid);
state0 = initResSol(G, 50*barsa, [0, 1]);
state0.wellSol = initWellSolAD([], model, state0);
```

Notice that the AD-OO framework assumes that the state object contains a subfield holding well solutions. Since there are no wells in our case, we instantiate this field to be an empty structure having the correct subfields. Finally, we set up the correct drive mechanisms: constant rate at the left and constant pressure at the right end.

```
injR = -sum(poreVolume(G,rock))/(500*day);
bc   = fluxside([], G, 'xmin', -injR, 'sat', [1, 0]);
bc   = pside(bc, G, 'xmax', 0*barsa, 'sat', [0, 1]);
```

Having set up the problem, the next step is to instantiate the nonlinear solver:

```
solver = NonLinearSolver();
```

The default setting means that MATLAB’s standard backslash operator is used to solve the linear problems. The time-step selector employs the simple time-chop strategy outlined in Section 10.2.2 to run the specified time steps; that is, time steps are chopped in two if the Newton solver requires more than 25 iterations, and increased back by a factor 50% after five successful chopped steps. The setup is the same as for the SPE 1 test case on page 427, except that we then used inexact line search to stabilize the Newton updates. Line search is not needed here since the fluid behavior of the Buckley–Leverett problem is much simpler and does not involve transitions between undersaturated and saturated states like in the SPE 1 case.

There are several ways we can use the nonlinear solver class. One alternative is to manually set up the time loop and only use the `solveTimestep` function of the nonlinear solver to perform one iteration:

```
[dT, n] = deal(20*day, 25);
states = cell(n+1, 1);
states{1} = state0;
for i = 1:n
    states{i+1} = solver.solveTimestep(states{i}, dT, model, 'bc', bc);
end
```

The results are stored in a generic format and can be plotted with the `plotToolbar` GUI discussed in the previous chapter. Alternatively, we can construct a simple simulation schedule and call the generic simulation loop implemented in `ad-core`:

```
schedule = simpleSchedule(repmat(dT,1,25), 'bc', bc);
[~,sstates] = simulateScheduleAD(state0, model, schedule);
```

This loop has a optional input argument that allows for dynamic plotting after each step in the simulation, for instance to show how the well curves progress during the simulation, or to print out extra information to the command window. Here, we use this hook to set up a simple graphical user interface that visualizes the progress of the simulation and enables you to modify the time steps interactively, dump the solution to workspace, and stop the simulation and continue running it in 'debug' mode. The GUI may in turn call any or both of the two GUIs shown in Figure 11.32 to plot reservoir states and well solutions. We request 1D plotting of the water saturation and turn off plotting of wells since the example does not have any:

```
fn = getPlotAfterStep(state0, model, schedule, ...
    'plotWell', false, 'plotReservoir', true, 'field', 's:1', ...
    'lockCaxis',true, 'plot1d', true);

[~,sstates,report] = ...
    simulateScheduleAD(state0, model, schedule,'afterStepFn', fn);
```

Figure 12.3 shows a snapshot of the GUI and the saturation profile just before the displacement front reaches the right end of the domain. The first control step required 7 iterations, whereas the next 15 steps only used 4 iterations, giving a total of 67 iterations or an average of 4.19 iterations for the first 16 control steps. The GUI also reports the total run time so far, and uses the iteration history to estimate the total runtime. A total runtime of seven seconds for such a small model may seem high, and can mainly be attributed to the GUI, plotting, and computational overhead introduced by the AD-OO framework. This overhead is significant for small models, but decreases in importance as the size of the model increases.

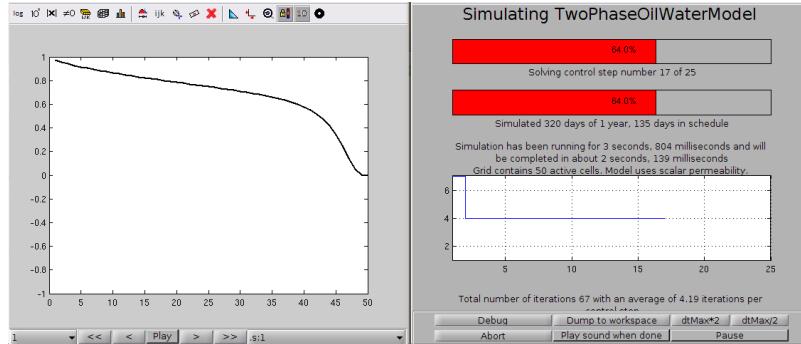


Fig. 12.3. The default graphical user interface in the generic simulation loop reports progress of simulation and enables simple computational steering.

For large models it may not be feasible to run interactive sessions and store all simulation results in memory. The solver also has a hook that enables you to control how the computed states are stored. To control the storage, we use a so-called storage handler, which behaves almost like a cell array, whose content can either be stored in memory, on disk, or both places. This is very useful if you have long simulations you want to run in batch mode, or if you expect that the simulator may fail or be interrupted before the simulation is finished and you want to store all completed time steps. Let us set up a handler that for storing computed states to disk

```
handler = ResultHandler('writeToDisk', true, 'dataFolder', 'AD-BL1D');
```

We can then pass the handler to the simulator as an optional argument

```
simulateScheduleAD(state0, model, schedule, 'outputHandler', handler);
```

The simulator will now store each state as a separate MATLAB formatted binary file on in a subdirectory 'AD-BL1D' somewhere on a standard path relative to where you have installed MRST. By default, the files are called `state1.mat`, and so on. The name can be changed through the option 'dataPrefix', whereas the parameter 'dataDirectory' lets you change the base directory relative to which the files are stored. To maintain results in memory, you need to enable the option 'storeInMemory'.

The result handler can now be used as a standard cell array, and you need not really know where the results are stored. The results can for instance be plotted through the same GUI we used above

```
figure
plotToolbar(G, handler, 'field', 's:1', 'lockCaxis', true, 'plot1d', true);
```

If you at a later time wish to retrieve the data, you can create a new handler and will then have access to all data stored on disk. The following example shows how to extract every second time step:

```

handler = ResultHandler('dataFolder', 'AD-BL1D');
m = handler.numelData();
states = cell(m, 1);
for i = 1:2:m
    states{i} = handler{i};
end

```

The stored data can be deleted through the call `handler.resetData()`.

To make the AD-OO simulator framework as versatile as possible, we have tried to develop a modular design, so that any simulator consists of a number of individual components that can easily be modified or replaced if necessary. Figure 12.4 shows how various classes, structures, and functions work together in the generic `simulateScheduleAD` function. The function takes physical model, initial state, and schedule as input, and then loops through all the control steps, updating time steps and drive mechanisms, calling the nonlinear solver, and performing any updates necessary after the nonlinear solver has converged (to track hysteretic behavior, etc). In the next two sections, we will discuss the main elements shown in Figure 12.4.

12.2 Model hierarchy

The `mrst-core` module implements two basic model classes `PhysicalModel` and `ReservoirModel`. Their main purpose is to provide generic templates for a wide variety of specific flow models. As such, these two generic model classes do not offer any flow equations and hence cannot be used directly for simulation, but provide generic quantities and access mechanisms, which we will discuss in more detail shortly. From these generic models, one can develop specific models by expanding the generic models by specific flow equations and fluid and rock properties. The module also contains classes implementing well models, both simple Peaceman wells of the type we discussed in and more advanced multisegment well models that give more accurate representation of deviated, horizontal, and multilateral wells with advanced completions and inflow control devices.

Compressible black-oil models of the type discussed in the previous chapter are implemented in the `ad-blackoil` module, which utilizes functionality from the `deckformat` and `ad-props` modules to read ECLIPSE input decks and construct models for rock-fluid and PVT properties, see Figure 12.5. The resulting simulators can simulate industry-standard black-oil models and compute adjoint gradients and sensitivities. Extensions of the black-oil equations to water-based EOR methods like polymer and surfactant flooding are implemented in the `ad-eor` module; see [26] for more details.

Sequential solvers, like those discussed in Chapter 10 can often be significantly faster than fully implicit solvers, especially for where the total velocity changes slowly during the simulation. Such solvers are also a natural starting point when developing e.g., multiscale methods [179, 180]. The

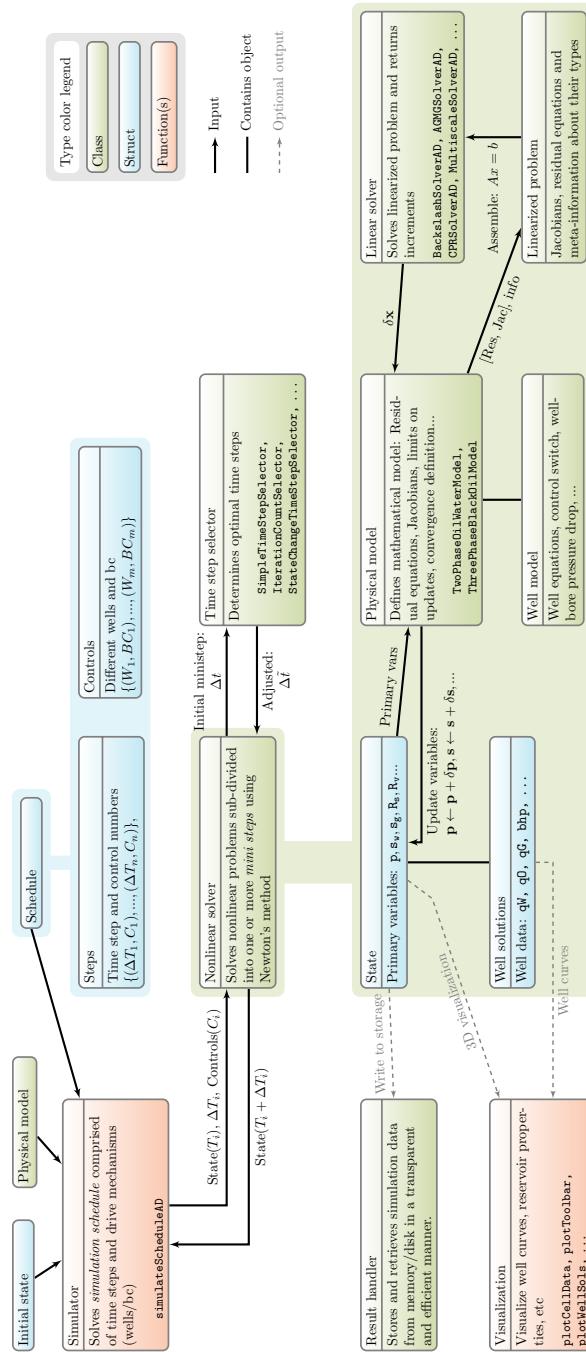


Fig. 12.4. Overview of how components in the AD-OO framework are organized to implement a generic simulator loop (here shown for black-oil equations).

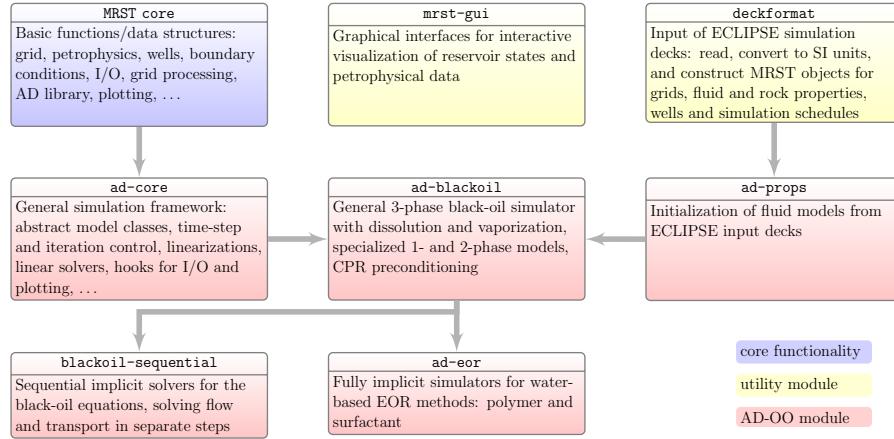


Fig. 12.5. Modules used to implement black-oil models in the AD-OO framework.

blackoil-sequential module implements sequential (fully implicit) solvers for the same set of equations as in **ad-blackoil**, based on a fractional flow formulation wherein pressure and transport are solved as separate steps and thus are represented as separate models. Last, but not least, the **co2lab** module uses inheritance from the **black-oil** module to implement specialized simulators for CO₂ sequestration. **MRST** also includes a family of *compositional* simulators as well as special solvers for geomechanics and for coupled flow and geomechanics. These are all outside the scope of this book; more information of the EOR and compositional solvers can be found in [26] and [176], respectively.

12.2.1 PhysicalModel – generic physical models

At the lowest level of the model hierarchy we have a generic physical model that defines basic properties to be inherited by other models and model classes. All models in the AD-OO framework are interpreted in a discrete sense and are assumed to consist of a set of model equations on residual form. The primary properties of a physical model are a structure holding *discrete operators* used to define the model equations and a *nonlinear tolerance* that defines how close the residual values must be to zero before the model is fulfilled. Actual operators are *not* implemented in the **PhysicalModel** class and must be specified in derived classes or explicitly set by the user for concrete instances of **PhysicalModel**. These operators are typically defined over a *grid*, and the **ad-core** module offers the function **setupOperatorsTPFA** for setting up the two-point operators we have discussed in previous chapters. In addition to the optional *grid* property, the class contains a flag informing if the model equations are linear (and hence can be solved by a single matrix inversion), and a flag determining how verbose the member functions will be in their output

```

classdef PhysicalModel
properties
    operators
    nonlinearTolerance
    G
    verbose
    stepFunctionIsLinear
end
methods
    function model = PhysicalModel(G, varargin)
        model.nonlinearTolerance = 1e-6;
        model.verbose = mrstVerbose();
        model = merge_options(model, varargin{:});
        model.G = G;
        model.stepFunctionIsLinear = false;
    end
end

```

Notice that all class names in MRST start with a capital letter to distinguish them from functions, which always start with a lower-case letter.

Model equations and physical properties

The basic class has several member functions implementing generic tasks associated with a model. The first, and most important, is to evaluate residual equations and Jacobians, which is done by the following function:

```
[problem, state] = model.getEquations(state0, state, dt, forces, vararg)
```

Here, `problem` is an instance of the `LinearizedProblem` class that implements what we referred to as the linearization numerical context in the simulator loop in Figure 12.1. This class has data structures to represent the linearized system for given states and member functions that correctly assemble the full linear system from residuals and Jacobian block matrices stored for the individual sub-equations making up the flow model; we will come back to more details in Section 12.3. Input argument `forces` represents the driving forces active at the current instance in time. These may vary from one step to the next and are specified by the control part of the schedule; see e.g., Figure 12.4. Active driving forces can be set by

```
forces = model.getDrivingForces(control);
```

which will create a cell array listing the forces on the same form as for the incompressible solvers. What are the valid forces for a model (with reasonable defaults) can be queried

```
validForces = model.getDrivingForces()
```

In addition, the class has utility functions for interacting with the `state` structure in a consistent manner. This includes querying the values of one or more specific variables/properties, setting or capping specific values, or correctly incrementing values

```
p      = model.getProp (state, 'pressure');
[p,s] = model.getProps(state, 'pressure', 's');
state = model.setProp (state, 'pressure', 5);
state = capProperty (state, 's', 0, 1)
state = model.incrementProp(state, 'pressure', 1);
```

These operations are generic and hence implemented in the `PhysicalModel` class. However, the specific examples above will only work for instances of `ReservoirModel` or other derived classes that specify the variables `pressure` and `s`. Checking whether a variable is defined or not, can be done by

```
[fn, index] = model.getVariableField(name)
```

which produces the correct field name and (sub)index that will extract the correct variable from the state object. This function will produce an error when called with any name for a `PhysicalModel` since this class does not implement any variables.

Linear updates to states

The second purpose of the basic model class is to correctly update states as part of a linear or nonlinear solution process. First of all, the function

```
[state, report] = model.stepFunction(state, state0, dt, forces, ...
linsolver, nonlinsolver, itno, varargin)
```

implements the generic behavior of a single nonlinear update, that is, executes the steps of the inner `while` loop of the right-most box in Figure 12.1. Here, `nonlinsolver` is an instance of the `NonLinearSolver` class, which we briefly introduced on page 427, and `linsolver` is an instance of a similar *linear* solver class; more detail are given in Section 12.3. The second output, `report`, is a standardized step report giving information like the CPU time used by the linear solver, setup of the stabilization step, convergence status, size of residual, etc. There are also functions to extract the computed update, update the state, check whether it is converged or not, check whether the controls have changed, as well as a hook that enables a state to be updated after it has converged (e.g., to model hysteretic behavior).

Let us in particular look at the function for updating the state using with a given increment. The following code will result in pressure being set to 110

```
state = struct('pressure', 10);
state = model.updateStateFromIncrement(state, 100, problem, 'pressure')
```

We can also restrict the maximum relative changes to be at most 10%

```
state = model.updateStateFromIncrement(state, 100, problem, 'pressure', .1)
```

which sets pressure to 11. Relative limits such as these are important when working with tabulated and nonsmooth properties in a Newton-type loop, as the initial updates may be far outside the reasonable region of linearization for a complex problem. On the other hand, limiting the relative updates can delay convergence for smooth problems with analytic properties and will, in particular, prevent zero states from being updated, so use with care.

Quality assurance

A model must also be able to check its own validity. The function

```
model.validateModel(forces)
```

where **forces** is an optional argument, validates that a model is suitable for simulation. If missing or inconsistent parameters can be fixed automatically, an updated model will be returned. Otherwise, an error should occur. This function does not have meaning for a **PhysicalModel**, but should be implemented in derived classes. Likewise,

```
model.validateState(state)
```

validates the state for use with the model and should check that required fields are present and of the right dimensions. If missing fields can be assigned default values, **state** is returned with the required fields added. If reasonable default values cannot be assigned, a descriptive error should be thrown telling the user what is missing or wrong (and ideally how to fix it). Also this function must be implemented in derived classes. The only function whose behavior *is* implemented in **PhysicalModel** is,

```
model.checkProperty(state, 'Pressure', [nc, 1], [1, 2]);
```

which in this particular call checks that the pressure field has **nc** elements along its first dimension and one element along the second dimension.

Adjoint equations

One of the reasons for introducing the AD-OO framework was to simplify the solution of so-called adjoint equations to compute parameter gradients and sensitivities (details of which are outside the scope of this book). The base class therefore contains an interface for computing adjoint equations,

```
[problem, state] = model.getAdjointEquations(state0,state,dt,forces,vararg)
```

Solving the adjoint equations essentially boils down to running a backward simulation with a set of linearized equations, and all this functions does by default is to call the **getEquations** function with the flag **reverseMode** set to true. The **PhysicalModel** class also offers a generic update function

```
[gradient, result, report] = model.solveAdjoint(..)
```

that computes a single step for the corresponding adjoint equations.

12.2.2 ReservoirModel – basic reservoir models

The next class in the hierarchy has properties to represent entities found in most reservoir models: fluid model, petrophysical properties, indicators for each of the basic three phases (aqueous, gaseous, and oleic), variables representing phase saturations and chemical components, and a model of production facilities

```
classdef ReservoirModel < PhysicalModel
properties
    fluid, rock, gravity, FacilityModel
    water, gas, oil
    saturationVarNames, componentVarNames

    % Iterations parameters
    dpMaxRel, dpMaxAbs, dsMaxRel, dsMaxAbs, maximumPressure, minimumPressure,
    useCNVConvergence, toleranceCNV, toleranceMB
    :
```

We notice in particular that the reservoir model declares absolute and relative tolerances on how much pressures and saturations can change during an iteration step as well as lower and upper bounds on the pressure. In addition, the class declares that convergence of nonlinear iterations can be measured in two different ways, either by mass balance (MB) or by scaled residuals (CNV); more details will follow in Section 12.3.

To construct an instance of the class, we first construct an instance of a `PhysicalModel` and then proceed to declare the additional properties that are part of a reservoir model. The following is a somewhat simplified excerpt:

```
function model = ReservoirModel(G, varargin)
    model = model@PhysicalModel(G);

    % We are being called in format ReservoirModel(G, rock, fluid, ...)
    model.rock = varargin{1}; model.fluid = varargin{2};

    [model.water, model.gas, model.oil] = deal(false);

    model.dpMaxRel = inf;    model.dpMaxAbs = inf;
    model.dsMaxAbs = .2;    model.dsMaxRel = inf;
    :
    model.operators = setupOperatorsTPFA(G, model.rock, ..)
end
```

By default, the reservoir model class does not contain any active phases; these must be specified in derived classes. However, we do declare that discrete operators should be of the two-point type. We also declare limits on how much pressures and saturations may change from one iteration step to the next: the default is that changes in pressure can be arbitrarily large, whereas saturations are only allowed to change 20%.

Physical variables and active phases

Physical variables known to a particular class are declared by the following function:

```
function [fn,ix] = getVariableField(model, name)
    switch(lower(name))
        case {'pressure' , 'p'}
            ix = 1; fn = 'pressure' ;
        case {'s', 'sat', 'saturation'}
            ix = ':'; fn = 's' ;
        case {'so' , 'oil'}
            ix = find(strncmp(model.getSaturationVarNames, name));
            fn = 's' ;
        :
    end
```

The purpose of this function is to provide a convenient and uniform way to access physical variables without knowing their actual storage in the state object. As an example, let us consider saturation, which we from previous chapters know is stored in `state.s`. However, this vector may have one, two, or three entries per cell depending upon the number of phases present. To access individual phase saturations, we introduce two additional utility functions:

```
function vars = getSaturationVarNames(model)
    vars = {'sw', 'so', 'sg'};
    vars = vars(model.getActivePhases());
end
function isActive = getActivePhases(model)
    isActive = [model.water, model.oil, model.gas];
end
```

Here, we see that oil saturation are stored in the second column of `state.s` for models containing water, but in the first column for a model without water. Oil saturation can now be extracted as follows:

```
[fn, ix] = model.getVariableField('so');
so = state.(fn)(:, ix);
```

This may seem unnecessary complicated, but enables access to variables in a uniform way for all models so that our code continues to work if we later were to implement models with more than three phases to represent precipitated solids, emulsified phases, etc.

Altogether, the reservoir model declares the possible presence of variables for pressure, saturations for the aqueous, oleic, and gaseous phases, temperature, and well solutions, but does not declare variables for dissolved gas-oil ratio and vaporized oil-gas ratios, as these are specific to black-oil models. There are also a number of other functions for getting the active phases and components and correctly storing saturations, mobilities, face fluxes, densities, shrinkage factors, and upstream indices in the state object.

Fluid behavior

The `ReservoirModel` class implements a generic method for evaluating relative permeabilities

```
function varargout = evaluateRelPerm(model, sat, varargin)
    active = model.getActivePhases();
    nph = sum(active);
    varargout = cell(1, nph);
    names = model.getPhaseNames();

    if nph > 1
        fn = ['relPerm', names];
        [varargout{:}] = model.(fn)(sat{:}, model.fluid, varargin{:});
    elseif nph == 1
        varargout{1} = model.fluid.(['kr', names])(sat{:}, varargin{:});
    end
```

The function is a general interface to the `relPermW0`, `relPermOG`, `relPermWG`, and `relPermWOG` functions discussed in Section 11.2, which in turn are interfaces to the more basic `krW`, `kr0`, `krG`, `kr0G`, and `kr0W` functions of fluid objects in MRST. The generic reservoir model neither implements nor offers any interface for PVT behavior since this will be very different in black-oil models and models relying on an equation of state. However, there is a function for querying surface densities of all phases.

Global driving forces

The basic physical model was aware of driving forces, but it did not implement any. The reservoir model has gravity as a property and can evaluate the gravity vector and compute a gravity gradient. In addition, the class specifies the potential presence of boundary conditions, source terms, and wells

```
function forces = getValidDrivingForces(model)
    forces = getValidDrivingForces@PhysicalModel(model);
    forces.W  = [];
    forces.bc = [];
    forces.src = [];
end
```

Wells are special since each well will be represented by another model, i.e., an instance of a derived class of `PhysicalModel`. Instances of such models are stored in the `FacilityModel` property of the reservoir model, which contains models of the production facilities used to bring hydrocarbons from the reservoir to the surface. New well instances are created by the following function whenever the well controls change

```
function [model, state] = updateForChangedControls(model, state, forces)
    model.FacilityModel = model.FacilityModel.setupWells(forces.W);
    state.wellSol = initWellSolAD(forces.W, model, state);
    [model,state] = updateForChangedControls@PhysicalModel(model, state, forces);
end
```

Adding in the effect of wells to a system of equations by adding the corresponding source terms and augmenting the system with additional model equations for the wells is a generic operation, which is done in the `insertWellEquations` function. The same holds for source terms and boundary conditions. The interested reader can consult the code for further details.

Updating states and model

Computing linearized increments in a reservoir model follows the exact same pattern as in a generic Newton loop and can hence be inherited from the `PhysicalModel` class without any special adaptions or extensions. On the other hand, the way these increments are used to update reservoir states will differ from other types of models and hence requires special implementation. The function `updateState` splits state variables into four different categories: well variables, saturation variables, pressure, and optional remaining variables. Updating well variables is quite involved and will not be discussed herein; as always, you can consult the code to find the necessary details.

Saturation variables are characterized by the fact that they should sum to unity so that pore space is completely filled by fluid phases. If our model contains n_{ph} phases, the linearized increments will contain $n_{ph} - 1$ saturation increments. The last increment is set as the negative sum of all the other increments; i.e., values added to the active variables must be subtracted from the last inactive phase so that the total increment over the n_{ph} phases is zero. The increments are then passed to the `updateStateFromIncrement` function to compute new saturations satisfying limits on absolute and relative updates. To ensure physically correct states, the updated saturations are then cropped to the unit interval and then we renormalize saturations in any cells containing capped values so that $S_\alpha \leftarrow S_\alpha / \sum_\alpha S_\alpha$.

Pressure is updated by first updating according to relative/absolute changes and then capping values to fulfill limits on minimum and maximum pressure. Any remaining variables are updated with no limits on changes or capping.

Validating model and state

As you may recall from above, the physical model defined functions for validating models and states but did not implement any behavior. The reservoir model checks that the model instance contains a facility model, and if not, instantiates one such object with default values.

```
function model = validateModel(model, varargin)
    if isempty(model.FacilityModel)
        model.FacilityModel = FacilityModel(model);
    end
    if nargin > 1
        W = varargin{1}.W;
        model.FacilityModel = model.FacilityModel.setupWells(W);
    end
    model = validateModel@PhysicalModel(model, varargin{:});

```

The validation function can also be called with a structure containing drive mechanisms as argument, in which case the well description is extracted and passed on to the facility model so that this model can initialize itself properly. The function then calls the validation function inherited from its parent class (`PhysicalModel`), which in the current implementation does nothing. Because the reservoir model does not contain any concrete equations and parameters, further consistency checks of model equations and petrophysical, fluid, and PVT properties must be performed in derived classes or implemented explicitly by the user. However, the model should be able to check a reservoir state and make sure that this state is compatible with the model:

```
function state = validateState(model, state)
    % Check parent class
    state = validateState@PhysicalModel(model, state);
    active = model.getActivePhases();
    nPh = nnz(active);
    nc = model.G.cells.num;
    model.checkProperty(state, 'Pressure', [nc, 1], [1, 2]);
    if nPh > 1
        model.checkProperty(state, 'Saturation', [nc, nPh], [1, 2]);
    end
    state = model.FacilityModel.validateState(state);

```

That is, the state should contain one unknown pressure and the correct number of phase saturations in each cell. Once this is done, the facility model should check that the reservoir state contains the correct data structures for representing well states. Neither of these operations check whether the *actual values* represented in the reservoir state are physically meaningful.

12.2.3 Black-oil models

The `ThreePhaseBlackOilModel` class is the base class in the `ad-blackoil` module and is derived from the `ReservoirModel` class. This base class is a *concrete class* designed to represent a general compressible, three-phase, black-oil model with dissolved gas and vaporized oil. The model therefore declares two new properties, `disgas` and `vapoil`, to signify the presence of dissolved gas and vaporized oil, respectively. The corresponding data fields `rs` and `rv` are defined in the `getVariableField` function, and `validateState` is extended to check that the reservoir state contains a `rs` and/or a `rv` field if these effects are active in the model. The class also defines the maximum absolute/relative increments allowed for r_s and r_v . By default, the constructor will set up a dead-oil model that contains the aqueous, gaseous, and oleic phases, but has no dissolution or vaporization effects. At the moment, the only way to specify a different model is through an ECLIPSE-style input deck.

Unlike the generic classes from `ad-core`, `ThreePhaseBlackOilModel` implements concrete equations. The member function `getEquations` calls the `equationsBlackOil`, which generates linearized equations according to how the general black-oil model discussed in Chapter 11 is configured. We have discussed the essential code lines contained in this function in the previous chapter, but the actual code is structured somewhat differently and contains more conditionals and safeguards to ensure that discrete residuals are computed correctly for all combinations of drive mechanisms and possible special cases of the general model.

In addition to implementing flow equations, the class extends `updateState` to implement variable switching when the reservoir state changes between saturated and undersaturated states. There is also a new member function that computes correct scaling factors CPR-type preconditioners, which in short derive an elliptic pressure equation by summing the conservation equations and then uses this as a preconditioner for the full implicit system.

The `ad-blackoil` also implements two special cases of the general black-oil model: a single-phase water model and a two-phase oil-water model. Since most functionality is inherited from the general model, these special cases are implemented quite compactly:

```
classdef TwoPhaseOilWaterModel < ThreePhaseBlackOilModel
properties
end
methods
    function model = TwoPhaseOilWaterModel(G, rock, fluid, varargin)
        model = model@ThreePhaseBlackOilModel(G, rock, fluid, varargin{:});
        model.oil = true;
        model.gas = false;
        model.water = true;
        model.saturationVarNames = {'sw', 'so'};
        model = merge_options(model);
    end

```

Evaluation of residual equations is done by the function `equationsOilWater`, which is a simplified version of the `equationsBlackOil` function in which all effects relating to gas, dissolution, and vaporization have been removed.

```
function [problem, state] = ...
    getEquations(model, state0, state, dt, drivingForces, varargin)
    [problem, state] = ...
        equationsOilWater(state0, state, model, dt, drivingForces, varargin{:});
end
```

For completeness, let us quickly go through the essential parts of this function, disregarding source terms, boundary conditions, and computation of adjoints. We start by getting driving forces and discrete operators and setting names of our primary variables,

```
W = drivingForces.W;
s = model.operators;
primaryVars = {'pressure', 'sW', wellVarNames{:}};
```

Then, we can use the reservoir and facility models to extract values for the primary unknowns:

```
[p, sW, wellSol] = model.getProps(state, 'pressure', 'water', 'wellsol');
[p0, sW0, wellSol0] = model.getProps(state0, 'pressure', 'water', 'wellSol');
[wellVars, wellVarNames, wellMap] = ...
    model.FacilityModel.getAllPrimaryVariables(wellSol);
```

Once this is done, we compute the residual equations more or less exactly the same way as discussed in Section 11.5. That is, we first compute fluid and PVT properties in all cells. Then, we compute gradient of phase potentials, apply suitable multipliers, and average properties correctly at the interface to compute intercell fluxes.

```
s0 = 1 - sW;
[krW, kr0] = model.evaluateRelPerm({sW, s0});
[pvMult, transMult, mobMult, pvMult0] = getMultipliers(model.fluid, p, p0);
krW = mobMult.*krW; kr0 = mobMult.*kr0;
T = s.T.*transMult;
:
```

We now have all we need to compute the residual equations

```
water = (s.pv/dt).*( pvMult.*bW.*sW - pvMult0.*bW0.*sW0 ) + s.Div(bWvW);
oil   = (s.pv/dt).*( pvMult.*b0.*s0 - pvMult0.*b00.*s00 ) + s.Div(b0v0);
```

In all our previous codes, wells have been included by explicitly evaluating the well equations and adding the corresponding source terms to the residual equations in individual cells before concatenating all the residual equations. Here, however, we first concatenate the reservoir equations and then use a

generic member function from the `ReservoirModel` class to insert the necessary well equations

```
eqs = {water, oil};
names = {'water', 'oil'};
types = {'cell', 'cell'};
[eqs, names, types, state.wellSol] = ...
    model.insertWellEquations(eqs, names, types, wellSol0, wellSol,
        wellVars, wellMap, p, mob, rho, {}, {}, dt, opt);
```

Finally, we construct an instance of the linearized problem class

```
problem = LinearizedProblem(eqs, types, names, primaryVars, state, dt);
```

We will discuss this class in more detail in Section 12.3.1.

12.2.4 Models of wells and production facilities

The AD-OO framework has primarily been developed to study hydrocarbon recovery, CO₂ storage, geothermal energy, and other applications in which fluids are produced from a porous rock formation and brought up to the surface, or brought down from the surface and injected into a rock formation. By default, all models are therefore assumed to have an entity representing this flow communication between the surface and the subsurface. For cases describing flow in the subsurface only, this entity will be void, but must still be present.

Facility models

The well models discussed earlier in the book apply to a single well (see Section 11.6). However, wells will oftentimes be operated in groups that are subject to an overall control and hence the corresponding well models must be coupled. Likewise, wells can be coupled so that gas or water produced from one well is injected into another well as displacing fluid. Wells from different reservoirs may sometimes also produce into the same flowline or surface network and hence be subject to overall constraints that couple the reservoir models. Hence, there is a need to represent a wider class of *production and injection facilities*.

Instead of talking of a well model, the AD-OO framework therefore introduces a general class of *facility models* that can represent different kinds of facilities used to enhance and regulate the flow of reservoir fluids from the sandface to the well head. The `FacilityModel` class is derived from the `PhysicalModel` class and is a general container class that holds a collection of submodels representing individual wells. In principle, the `FacilityModel` container class can also incorporate models for various types of topside production facilities, but at the time of writing, no such model is implemented in MRST.

Peacemann-type well models

The inflow-performance relation, $q = J(p_R - p_{wb})$, discussed in Sections 4.3.2 and 11.6.1 gives an unsophisticated description of injection and production wells, and is generally best suited for simple vertical wells, in which fluids inside the wellbore can be assumed to be in hydrostatic equilibrium. Apart from the well index (and possibly a skin factor), no attempt is made to model the complex flow physics that may take place in various types of well completions. The flow in and out of the wellbore is regulated by controlling either the surface flow rate or the bottom-hole pressure. In the AD-OO framework, this type of well model is implemented in the `SimpleWell` class, which is derived from the `PhysicalModel` class.

Multisegment well models

The `MultisegmentWell` class is also derived from the `PhysicalModel` class and implements the more general concept of multisegment wells, which was briefly introduced in Section 11.6.2. Here, the flow of fluids from the sandface to the well head is described in terms of a set of 1D models on a general flow network. In ECLIPSE [226], the network needs to be in the form of a tree (i.e., be a directed graph). The implementation in MRST is more general and allows for complex network topologies involving circular dependencies among the nodes.

Details of the implementation

Implementing the correct behavior of wells and facilities involves quite complicated logic and a lot of intricate details that quickly make the code difficult to understand. A detailed discussion of the facility and well classes is therefore outside the scope of this book. As usual, the interested reader can find all necessary details in the actual code. However, let me add a small warning: To retain the same computational efficiency as the rest of the simulator classes, the well classes have been optimized for computational efficiency. The reason is that well and facility models typically have much fewer unknowns than what is associated with the reservoir itself. The vectorized AD library in MRST has been designed to perform well for long vectors and large sparse matrices, and to keep the computational efficiency when working with individual wells, it has been necessary to meticulously distinguish between variables that need automatic differentiation and variables that do not. As a result, the code contains explicit casting of AD variables to standard variables, which in my opinion has reduced readability.

In other words: please go ahead and read the code, but be warned that it may appear less comprehensible than other parts of the AD-OO framework.

12.3 Solving the discrete model equations

In this section, we will go through the three different types of classes the AD-OO framework uses to solve the discrete equations to motivate their design and briefly describe their key functionality. For those interested in developing their own solution algorithms and solvers, the classes define generic information about the discrete equations and interfaces to the rest of the simulator you can use as a starting point.

12.3.1 Assembly of linearized systems

The main purpose of the `LinearizedProblem` class is to assemble the individual matrix blocks computed by the AD library invoked within a model class into a sparse matrix representing the linearization of the whole reservoir model. To this end, the class contains the residual equations evaluated for a single state along with meta-information about the equations and the primary variables they are differentiated with respect to, which we will refer to as a *linearized problem*. This description can be transformed into a linear system and solved using a subclass of the `LinearSolverAD` class to be discussed in Section 12.3.4 provided that the number of equations matches the number of primary variables. The class also contains the following data members:

```
classdef LinearizedProblem
properties
    equations, types, equationNames
    primaryVariables
    A, b
    state
    dt, iterationNo, drivingForces
end
```

Here, `equations` is a cell array containing the residuals evaluated at the reservoir state represented in the `state` object. The residuals can be real numbers but are most typically AD objects. The cell array `types` has one string per equation indicating its type (common types: '`'cell'`' for cell variables, '`'well'`' for well equations, etc.). These types are available to any linear solver and can be used to construct appropriate preconditioners and solver strategies. The cell arrays `equationNames` and `primaryVariables` contains the names of equations and primary variable; these are used e.g., by the linear solvers and for reporting convergence (see below). The linear system is represented in the sparse matrix `A` and the right-hand side in the vector `b`. For completeness, we also store the time step, the iteration number, and the driving forces, which may not be relevant for all problems.

The linear system corresponding to a linearized problem can be extracted using the member function

```
[A, b] = problem.getLinearSystem();
```

The function will check whether the linear system already exists, and if not, it will be assembled using the function:

```
function problem = assembleSystem(problem)
    if isempty(problem.A)
        iseq = cellfun(@(x) ~isempty(x), problem.equations);
        eqs = combineEquations(problem.equations{iseq});
        if isa(eqs, 'ADI')
            problem.A = -eqs.jac{1};
            problem.b = eqs.val;
        else
            problem.b = eqs;
        end
    end
end
```

Here, the vertical concatenation of equations is implemented as a separate one-line function. If the residual equations are given as real numbers, this function reads

```
function h = combineEquations(varargin)
    h = vertcat(varargin{:});
end
```

i.e., we simply concatenate the vectors representing residuals of each individual equation into one long column vector representing the residuals for the whole system. In this case, we cannot construct a linear system since no partial derivatives are available. If the residual equations are represented as AD objects, the linear system is, as before, assembled from the Jacobian block matrices stored for each individual (continuous) equation. The AD library from `mrst-core` implements an overloaded version of `combineEquations` which ensures that AD objects are concatenated correctly and also enables concatenation of AD objects and standard doubles.

How the linear equations are solved, will obviously depend on the linear solver. The simplest approach is to solve all equations at once. However, as we will come back to in Section 12.3.4, it may sometimes be better to eliminate some of the variables and only solve for a subset of the primary variables. To this end, the `LinearizedProblem` class has member functions that use a block-Gaussian method to eliminate individual variables

```
[problem, eliminatedEquation] = problem.eliminateVariable(name)
```

where `name` corresponds to one of the entries in `problem.equationNames`. Likewise, one can eliminate all variables that are not of a specified type

```
[problem, eliminated] = reduceToSingleVariableType(problem, type)
```

This could, for instance, be all equations that are not posed on reservoir cells. There is also a function that enables you to recover the increments

in primary variables corresponding to equations that have previously been eliminated. In addition to the above, the `LinearizedProblem` class contains member functions for appending/prepending additional equations, reordering the equations, computing the norm of each residual equation, querying indices and the number of equations, querying the number of equations of a particular type, as well as a number of utility functions for sanity checks, clearing the linear system, and so on.

12.3.2 Nonlinear solvers

The `NonLinearSolver` class is based on a standard Newton–Raphson formulation and is capable of selecting time steps and cutting them based on convergence rate. To modify how the solver works, you can either develop your own subclass and/or combine the existing solver with modular linear solvers and classes for time-step selection. You have already seen some of the properties of the class in Section 11.7.1 on page 427. The following is an almost complete declaration of the class and its data members

```
classdef NonLinearSolver < handle
properties
    identifier
    maxIterations, minIterations
    LinearSolver, timeStepSelector, maxTimestepCuts
    useRelaxation, relaxationParameter, relaxationType, relaxationIncrement
    minRelaxation, maxRelaxation
    useLinesearch, linesearchReductionFn
    linesearchReductionFactor, linesearchDecreaseFactor, linesearchMaxIterations
    linesearchConvergenceNames, linesearchResidualScaling
    enforceResidualDecrease, stagnateTol
    errorOnFailure, continueOnFailure
```

The nonlinear solver needs to call a linear solver and hence has a class object; the default is the standard backslash solver. Likewise, we need time-step selection, which in the default does not offer any advanced heuristics for computing optimal time steps. Furthermore, there are limits on the number of times we are allowed to subdivide the control steps and the maximum/minimum number of iterations within each subdivided step. If `useRelaxation` is true, the computed Newton increments should be relaxed by a factor between 0 and 1. The `relaxationParameter` is modified dynamically if `useRelaxation` is true and should in general not be modified unless you know what you are doing. Relaxation can be any of the following three choices

```
x_new = x_old + dx; % relaxationType = 'none'
x_new = x_old + dx*w; % relaxationType = 'dampen'
x_new = x_old + dx*w + dx_prev*(1-w); % relaxationType = 'sor'
```

where `dx` is the Newton increment and `w` is the relaxation factor. Similarly, there are various parameters for invoking and controlling line search.

When `enforceResidualDecrease` is invoked, the solver will abort if the residual does not decay more than the stagnation tolerance during relaxation. If `errorOnFailure` is not enabled, the solver will return even though it did not converge. Although non-converged results should not be relied upon, this behavior may be useful for debugging purposes.

Use of handle classes

The careful reader may have observed that `NonLinearSolver` is implemented as a subclass of the `handle` class. All classes defined so far have been so-called value classes, meaning that whenever you copy an object to another variable or pass it to a function, MATLAB creates an independent copy of the object and all the data the object contains. This means that if you want to change data values of a class object inside a function, you need to return the class as an output variable. Classes representing nonlinear solvers do a lot of internal bookkeeping inside a simulator and hence need to be passed as argument to many functions. By making this class a *handle class*, the class object is passed by reference to functions, so that any changes occurring inside the function will also take effect outside the function without having to explicitly return the class object as output argument.

Computing a control step

We have already encountered the primary member function

```
[state, report, ministates] = solver.solveTimestep(state0, dT, model)
```

whose purpose is to solve a specified control step. Recall from our discussion of nonlinear Newton-solvers in Section 10.2.2 that each control step may be subdivided into several substeps (ministeps) to ensure stable computations and proper convergence. Each ministep will involve of one or more *nonlinear* iterations, which each may consist of one matrix inversion if we use a direct linear solver or one or more *linear iterations* if we use an iterative solver.

Whether the solver will move forwards using a single time step or multiple substeps depends on the convergence rate and what the time-step selector predicts to be a feasible time step for the given state. For the `solveTimestep` function to work, the `model` object must contain a valid implementation of the `stepFunction` member function declared in the `PhysicalModel` class. Driving forces that are active for the particular model must be passed as optional arguments in the same way as for the incompressible solvers. Upon completion, the function will return the reservoir state after the time step, a report structure containing standard information like iteration count, convergence status, and any other information passed on by `model.stepFunction`. The cell array `ministates` contains all ministeps used to advance the solution a total time step `dT`. The class also has member functions for computing these ministep, applying line search, stabilizing Newton increments, and implementing checks for stagnation or oscillating residuals. You have already

been exposed to the essential ideas of these operations; the implementation in the `NonLinearSolver` class contains more safeguards against possible errors, has more flexible hooks for plug-able linear solvers and time-step selection schemes, as well as more bells and whistles for monitoring and reporting the progress of the solver.

Measures of nonlinear convergence

Whether the nonlinear solution process has converged or not is determined by the `PhysicalModel` class; the `NonLinearSolver` class simply responds based on what the model reports in terms of convergence. There are several different ways convergence can be measured; the AD-OO framework currently implements the same two measures of convergence as in ECLIPSE [227]. The first measure is mass balance. When summing the residual equations over all cells in the grid, all intercell fluxes will cancel because the flux *out of* a cell across an internal face has the same magnitude but opposite sign of the corresponding flux *into* the neighboring cell. The sum of the residuals therefore equals the difference between the net mass accumulation of the phase minus the net influx from wells, source terms, and open boundaries, which is the definition of the mass-balance error. By convention, these errors are scaled as follows to make them problem-independent,

$$\text{MB}_\alpha = \Delta t \bar{B}_\alpha \left(\sum_i \mathcal{R}_{\alpha,i} / \sum_i \Phi_i \right), \quad (12.1)$$

where \bar{B}_α is the average formation-volume factor of phase α , $\mathcal{R}_{\alpha,i}$ is the residual equation evaluated in cell i , and Φ_i is the pore volume of that cell. The result can be interpreted as mapping from masses to saturations at field conditions. The solution process is considered as converged when the all scaled residuals are less than the `toleranceMB` tolerance declared in the `ReservoirModel` class. The second test computes the maximum normalized residual defined as

$$\text{CNV}_\alpha = \Delta t \bar{B}_\alpha \max_i \left| \frac{\mathcal{R}_{\alpha,i}}{\Phi_i} \right| \quad (12.2)$$

and says that the solution is converged when this quantity is less than `toleranceCNV` for all three phases. Default values for the MB and CNV tolerances are the same as in ECLIPSE 100, i.e., 10^{-7} and 10^{-3} , respectively.

12.3.3 Selection of time-steps

As indicated above, appropriate selection of time step is crucial to ensure low runtimes and good convergence in the nonlinear solver. We have already encountered several time-step selection schemes that use experience from previous time steps to try to select optimal time steps, e.g., by halving the time step if the nonlinear solver fails to converge, or increasing the time step back once we have managed to successfully run a certain number of chopped steps. Questions to be asked when determining the next time step include:

- How difficult was it to converge the previous time step? As a simple rule of thumb, a single iteration is usually considered as too easy, two to three iterations is easy, while more than ten iterations indicates problems with time step or model.
- For a control step: how does the specified step compare with the last successful step?
- For a ministep: how far are we away from the end of the control step?
- How does the required number of iterations for the last successful step compare with any iteration targets?
- Which mechanism determined the size of the previous time step?

Heuristics based on these and similar questions are implemented in the time-selector classes of MRST, which like `NonLinearSolver` are handle classes. The base class is defined as follows:

```
classdef SimpleTimeStepSelector < handle
properties
    history, maxHistoryLength
    isFirstStep, isStartOfCtrlStep
    previousControl, controlsChanged, resetOnControlsChanged
    firstRampupStep, firstRampupStepRelative
    maxTimestep, minTimestep
    maxRelativeAdjustment, minRelativeAdjustment
    stepLimitedByHardLimits
```

The base class implements a generic member function `pickTimestep`,

```
dt = selector.pickTimestep( dtPrev, dt, model, solver, statePrev, state)
```

which is called by the `NonLinearSolver` class to determine the next time step. The logic of this function is as follows: We start by checking whether this is the first step, or the well controls have changed and we want to induce a gradual ramp-up

```
if selector.controlsChanged && ...
    (selector.resetOnControlsChanged || selector.isFirstStep);
    dt = min(dt, selector.firstRampupStepRelative*dt);
    dt = min(dt, selector.firstRampupStep);
    selector.stepLimitedByHardLimits = true;
end
```

The default behavior is to not use such a ramp-up, which means that the default value of the relative adjustment is set to unity, the lower limit on the first ministep attempted after controls have changed is set to infinity, and the reset flag is disabled. The time step is then passed on to the function

```
dt = selector.computeTimestep( dt, dtPrev, model, solver, statePrev, state)
```

which will try to compute an optimal adjustment, using some sort of heuristics based on the stored time-step and iteration history (which by default consists

of 50 entries). The base class offers no heuristics and returns an unchanged time step. The `IterationCountTimeStepSelector` subclass tries to select a time step that will maintain the number of nonlinear iterations as close as possible to a prescribed target, whereas `StateChangeTimeStepSelector` is a time-step selector that attempts to ensure that certain properties of the state change at target rates during the simulation. This can often be a good way of controlling time steps and minimizing numerical error if good estimates of the error are known; see [50] for more details.

The last step of the selection algorithm is to ensure that the computed time step does not change too much compared with the previous time step unless we are at the start of a new control step. The default choice of tolerances is to allow relative changes of at most a factor two. Likewise, all time steps are required to be within certain lower and upper bounds (which by default are 0 and ∞ , respectively). Alternative heuristics for choosing time steps can be introduced by implementing a new subclass of the `SimpleTimeStepSelector` base class or one of its derived classes.

12.3.4 Linear solvers

The linear solver classes in AD-OO implement methods for solving linearized problems. All concrete solvers are subclasses of the following handle class:

```
classdef LinearSolverAD < handle
properties
    tolerance, maxIterations
    extraReport, verbose
    replaceNaN, replacementNaN, replaceInf, replacementInf
end
```

The class provides a general interface to all type of linear solvers, both direct and iterative, and hence the class declares both a tolerance for the linear solver (default: 10^{-8}) and an upper limit on the number of linear iterations (default: 25), even though these do not make sense to direct solvers. The flag `extraReport` determines the amount of reporting; turning it on may consume a lot of memory for problems with many unknowns.

The main interface to linear solvers is through the following member function, which assembles and solves a linear system:

```
function [dx, result, report] = solveLinearProblem(solver, problem, model)
    problem = problem.assembleSystem();
    :
    timer = tic();
    [result, report] = solver.solveLinearSystem(problem.A, problem.b);
    [result, report] = problem.processResultAfterSolve(result, report);
    report.SolverTime = toc(timer);
    :
    dx = solver.storeIncrements(problem, result);
```

The only place this function is called is within the generic `stepFunction` implemented in the `PhysicalModel` class. The code excerpts above show the main steps of solving a linear problem: First, the linear system is assembled by the `problem` object, which is an instance of the `LinearizedProblem` class discussed above. Then we call the main member function `solveLinearSystem`, which is nonfunctional in the base class and must be implemented in concrete subclasses, as discussed below. The linear solution is then passed back to the `LinearizedProblem` class for potential postprocessing. No such postprocessing is necessary for the models discussed herein, but the hook is provided for generality. Linear solvers may sometimes produce infinite values or not-a-number, which the linear solver class can replace by other values if the `replaceInf` and/or `replaceNaN` flags are set; this is done by two simple `if` statements, which are not shown for brevity. Finally, we extract the results from the solution vector and store them in the cell array `dx`, which has one increment entry per primary variable in the linearized problem.

In addition to the two member functions for solving linearized problems and linear systems, the `LinearSolverAD` class implements various utility functions that may be utilized by concrete linear solvers. This includes interfaces for setup and cleanup of linear solvers as well as a generic function that can reduce the problem by eliminating some of the variables and a similar generic function for recovering increments for eliminated variables from the linear solution. There is also special function that reduces a problem to cell-variables only, solves it, and then recovers the eliminated variables afterwards. Last, but not least, the `solveAdjointProblem` function solves backward problems for computing adjoints.

MATLAB's standard direct solver

By far, the simplest approach is to use the standard direct solver. Use of this solver is implemented very compactly:

```
classdef BackslashSolverAD < LinearSolverAD
    methods
        function [result, report] = solveLinearSystem(solver, A, b) %#ok
            result = A\b;
            report = struct();
        end
```

The resulting solver is very robust and should be able to solve almost any kind of linear system that has a solution. On the other hand, black-oil models have several unknowns per cell and can have many non-zero matrix entries because of unstructured grid topologies. Sparse direct solvers do generally not scale very well with the number of unknowns, which means that both the memory consumption and the runtime will increase rapidly with model sizes. MATLAB's direct solvers should therefore only be used for models having at most a few tens of thousand unknowns.

GMRES with ILU preconditioner

To get a more scalable solver, one needs to use an iterative method. Iterative methods used today to solve large-scale, sparse systems are usually of the Krylov subspace type, and seek to find an approximation to the solution of the linear system $Ax = b$ in the Krylov space formed by repeatedly applying the matrix A to the residual $r_0 = b - Ax_0$, where x_0 is some initial guess:

$$\mathcal{K}_\nu = \mathcal{K}_\nu(A, r_0) = \text{span}\{r_0, Ar_0, A^2r_0, \dots, A^{\nu-1}r_0\}.$$

One widely used example is the generalized minimal residual method (GMRES), which seeks the approximate solution as the vector x_ν that minimizes the norm of the ν 'th residual $\|r_\nu\| = \|b - Ax_\nu\|$. The Krylov vectors are often linearly dependent, and hence the GMRES method uses a modified Gram-Schmidt orthogonalization, called Arnoldi iteration, to find an orthonormal basis for \mathcal{K}_n .

The GMRES iteration is generally robust and work well for nonnsymmetric systems of linear equations. However, iterative solvers are only efficient if the condition number of the matrix is not too high. Black-oil models can often give linear systems having quite bad condition numbers. To remedy this, it is common to use a *preconditioner*. Here, we will apply an incomplete LU factorization (ILU) method. The LU factors for a typical sparse matrix can be much less sparse than the original matrix, and in an incomplete factorization, one seeks triangular matrices L and U such that $LU \approx A$. When used as a preconditioner, it is common to construct L and U so that these matrices preserve the sparsity structure of the original matrix, i.e., that we have no fill-in of new nonzero elements. The result is called ILU(0).

The `GMRES_ILUSolverAD` class implements a GMRES solver with ILU(0) preconditioning by use of the built-in `gmres` and `ilu` solvers from MATLAB. As for the direct solver, the implementation is quite compact:

```
function [result, report] = solveLinearSystem(solver, A, b)
    nel = size(A, 1);
    if solver.reorderEquations
        [A, b] = reorderForILU(A, b);
    end
    [L, U] = ilu(A, solver.getOptsILU());
    prec = @(x) U\ (L\x);
    [result, flag, res, its] = ...
        gmres(A, b, [], solver.tolerance, ...
            min(solver.maxIterations, nel), prec);
    report = struct('GMRESFlag', flag, ...
        'residual', res, 'iterations', its);
end
```

Notice that we permute the linear system to ensure nonzero diagonals to simplify the construction of the ILU preconditioner. Notably, this utility is

useful whenever well equations are added, since these may not have derivatives with respect to all well controls.

Well equations will generally consist of a mixture of different variables and hence be badly scaled. Control equation on pressure gives residuals having magnitudes that are typically $\mathcal{O}(10^7)$ since all equations are converted to SI units in MRST. Rate equations, on the other hand will give residuals that typically are $\mathcal{O}(10^{-5})$. To be on the safe side, we overload the solver for linearized problems by a new function that eliminates all equations that are not posed on grid cells by use of a block-Gaussian algorithm, solve the resulting problem by the GMRES-ILU(0) method, and then recover the eliminated variables

```
function [dx, result, report] = solveLinearProblem(solver, problem, model)
    keep = problem.indexOfType('cell');
    [problem, eliminated] = solver.reduceToVariable(problem, keep);
    problem = problem.assembleSystem();

    timer = tic();
    [result, report] = solver.solveLinearSystem(problem.A, problem.b);
    report.SolverTime = toc(timer);

    dxCell = solver.storeIncrements(problem, result);
    dx = solver.recoverResult(dxCell, eliminated, keep);
end
```

Elimination of well equations (and other non-cell equations) is done using member functions from the `LinearSolverAD` base class, which in turn relies on block-Gaussian algorithms implemented in the `LinearizedProblem` class. This algorithm is not specific to the GMRES-ILU(0) and in the actual code the content of this function is implemented as a member function of the `LinearSolverAD` class.

Constrained pressure residual (CPR)

The state-of-the-art approach for solving linear systems arising from black-oil type models is to use a so-called constrained pressure residual (CPR) method [253, 95]. To explain the method, we start by writing the equation for the Newton update in block-matrix form

$$-\begin{bmatrix} \mathbf{J}_{pp} & \mathbf{J}_{ps} \\ \mathbf{J}_{sp} & \mathbf{J}_{ss} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_p \\ \Delta\mathbf{x}_s \end{bmatrix} = \begin{bmatrix} \mathbf{R}_p \\ \mathbf{R}_s \end{bmatrix},$$

where we have decomposed the unknown Newton increment in primary unknowns (p), which typically consist of pressure variables, and secondary unknowns (s) that consist of saturations and mass fractions. The Jacobi matrix blocks read $(\mathbf{J}_{m,\ell})_{i,j} = \partial\mathbf{R}_{\ell,i}/\partial\mathbf{x}_{m,j}$, where the indices m, ℓ run over the variable types $\{p, s\}$ and i, j run over all cells.

Instead of applying an iterative solver to the full linear system, we first form an (almost) elliptic equation for the primary unknowns. This equation can be obtained by decoupling the fully implicit system into

$$-\begin{bmatrix} \mathbf{J}_{pp}^* & \mathbf{J}_{ps}^* \\ \mathbf{J}_{sp} & \mathbf{J}_{ss} \end{bmatrix} \begin{bmatrix} \Delta \mathbf{x}_p \\ \Delta \mathbf{x}_s \end{bmatrix} = \begin{bmatrix} \mathbf{R}_p^* \\ \mathbf{R}_s \end{bmatrix},$$

where the off-diagonals of the coupling matrix \mathbf{J}_{ps}^* are small, so that $\mathbf{J}_{pp}^* \Delta \mathbf{x}_p = \mathbf{R}_p^*$ resembles an incompressible pressure equation. Because this CPR pressure equation is almost elliptic, we can use one of the many efficient solvers developed for Poisson-type equations to compute an approximate increment $\Delta \mathbf{x}_p$ in an inexpensive manner. This *predictor stage* is used in combination with a *corrector stage*, in which we iterate on the entire system using the approximated pressure update as our initial guess.

The solver implemented in the `CPRSolverAD` class uses an IMPES-like reduction, in which one determines appropriate weights w_c so that the accumulation terms of the residual equations cancel

$$\sum_c w_c \mathcal{A}_c = 0, \quad \mathcal{A}_{c,i} = \left(\phi \sum_{\alpha} b_{\alpha} S_{\alpha} x_{c,\alpha} \right)_i^{n+1} - \left(\phi \sum_{\alpha} b_{\alpha} S_{\alpha} x_{c,\alpha} \right)_i^n.$$

Here, $x_{c,\alpha}$ gives the volume of component c at standard conditions present in a unit stock-tank of phase α at reservoir conditions, i.e., $x_{o,o} = 1$, $x_{o,g} = R_v$, and so on. The (almost) decoupled CPR pressure equation is then taken to be $\mathbf{R}_p^* = \sum_c w_c \mathbf{R}_c$. Notice that if components are in one phase only, the weight of component c is simply $1/b_{\alpha}$, where α is the phase of component c . An advantage of this decoupling strategy is that the resulting pressure block \mathbf{J}_{pp}^* is fairly symmetric. Moreover, the decoupling can be done at a nonlinear level, before the Jacobians are computed, and is thus less prone to errors.

Aggregation-based algebraic multigrid

The pressure equation in the CPR solver can obviously be solved using a standard direct solver. However, a more scalable approach is to use an iterative multilevel method. To present the idea, we consider two levels only: the fine level on which we seek the unknown solution \mathbf{x} and a coarser level on which we seek another unknown vector $\tilde{\mathbf{x}}$ having much fewer elements. To move between the two levels, we have a prolongation operator \mathbf{P} and a restriction operator \mathbf{Q} . Then the iterative method can be written as follows:

$$\begin{aligned} \mathbf{x}^* &= \mathbf{x}_{\nu} + \mathbf{S}(\mathbf{b} - \mathbf{A}\mathbf{x}_{\nu}) \\ \mathbf{x}_{\nu+1} &= \mathbf{x}^* + \mathbf{P}\mathbf{A}^{-1}\mathbf{Q}(\mathbf{b} - \mathbf{A}\mathbf{x}^*) \end{aligned}$$

Here, \mathbf{S} is a so-called smoother, which typically consists of a few steps of an inexpensive iterative solver (like Jacobi's method). For a multilevel method, this procedure is performed recursively on sequence of coarser levels. In classical methods, the coarser levels are chosen by geometric partitions of the original

grid. The state-of-the-art approach, however is to select the coarse levels algebraically and build the prolongation operator \mathbf{P} by analyzing the matrix coefficients of \mathbf{A} . The restriction operator is then chosen as the transpose of the interpolation matrix, $\mathbf{Q} = \mathbf{P}^T$, and the coarse matrix is $\mathbf{A}_c = \mathbf{P}^T \mathbf{A} \mathbf{P}$.

The `AGMGSolverAD` class is an interface to the aggregation-based algebraic multigrid (AMG) solver developed by Notay [203, 10]. The key idea of this solver is to build prolongation operators using recursive aggregates of cells so that the operator on each level is as sparse as possible to avoid that the coarse matrices \mathbf{A}_c from becoming more dense the more one coarsens. Earlier versions of AMG were released as free open-source, but the most recent versions are only freely available for academic users. The main advantage AMG, beyond a simple MATLAB interface, is that it has low setup cost and low memory requirements and in my experience, it seems to work very well with MRST. For the black-oil equations, this solver should either be used as a preconditioner for the CPR solver, or as a pressure solver for the sequentially implicit method implemented in the `blackoil-sequential` module.

As with many other multigrid solvers, AMG can be run in two stages: a setup stage that builds the coarse hierarchies and the prolongation operators, and a solution stage that uses the prolongation operators to solve a specific problem. One can then hope to reduce computational costs by reusing the setup stage for multiple subsequent solves. Setting up this behavior can be done as follows

```

function [result, report] = solveLinearSystem(solver, A, b)
    cleanAfter = false;
    if ~solver.setupDone
        solver.setupSolver(A, b); cleanAfter = true;
    end
    if solver.reuseSetup
        fn = @(A, b) agmg(A, b, [], solver.tolerance, ...
                           solver.maxIterations, [], [], 1);
    else
        fn = @(A, b) agmg(A, b, [], solver.tolerance, solver.maxIterations);
    end
    [result, flag, relres, iter, resvec] = fn(A, b);

    if cleanAfter, solver.cleanupSolver(A, b); end
end
function solver = setupSolver(solver, A, b, varargin)
    if solver.reuseSetup
        agmg(A, [], [], [], [], [], 1); solver.setupDone = true;
    end
end

```

For brevity, we have dropped a few lines that create the full convergence report. This example hopefully shows that it is not very difficult to set up an external solver to work with MRST if the solver has a standard MATLAB interface.

12.4 Simulation examples

You have now been introduced to all key components making up the AD-OO framework. In the following, we will demonstrate in more detail how to use the AD-OO framework to perform simulations with or without the use of structured input decks on the ECLIPSE format.

12.4.1 Depletion of a closed/open compartment

The purpose of this example¹ is to demonstrate how to set up a simulator from scratch without the use of input files. As an example we consider a 2D rectangular reservoir compartment with homogeneous properties filled by a single-phase fluid. The reservoir fluid is drained by a single producer at the midpoint of the top edge. The compartment is either closed (i.e., sealed by no-flow boundary conditions along all edges), or open with constant pressure support from an underlying, infinite aquifer (which we model as a constant-pressure boundary condition).

We start by making the geological model and fluid properties

```
G = computeGeometry(cartGrid([50 50], [1000 100]));
rock = makeRock(G, 100*milli*darcy, 0.3);

pR = 200*barsa;
fluid = initSimpleADIFluid('phases', 'W', ... % Fluid phase: water
                           'mu', 1*centi*poise, ... % Viscosity
                           'rho', 1000, ... % Surface density [kg/m^3]
                           'c', 1e-4/barsa, ... % Fluid compressibility
                           'cR', 1e-5/barsa ... % Rock compressibility
                           );
```

This is all we need to set up a single-phase reservoir model. To this end, we will use a `WaterModel`, which is a specialization of the general `ReservoirModel` implemented in `ad-core`. The only extra thing we need to do is to explicitly set the gravity direction. By default, the gravity in MRST is a three-component vector that points in the positive z -direction. Here, we set it to a two-component vector pointing in the negative y -direction.

```
wModel = WaterModel(G, rock, fluid, 'gravity', [0 -norm(gravity)]);
```

As always, we also need to specify the drive mechanisms by use of the standard data structures for wells and boundary conditions from `mrst-core`:

```
wc = sub2ind(G.cartDims, floor(G.cartDims(1)/2), G.cartDims(2));
W = addWell([], G, rock, wc, ...
            'Type', 'bhp', 'Val', pR - 50*barsa, ...
            'Radius', 0.1, 'Name', 'P1', 'Comp_i', 1, 'sign', 1);
bc = pside([], G, 'South', 200*barsa, 'sat', 1);
```

¹ Complete code: `blackoilTutorialOnePhase.m` in `ad-blackoil`

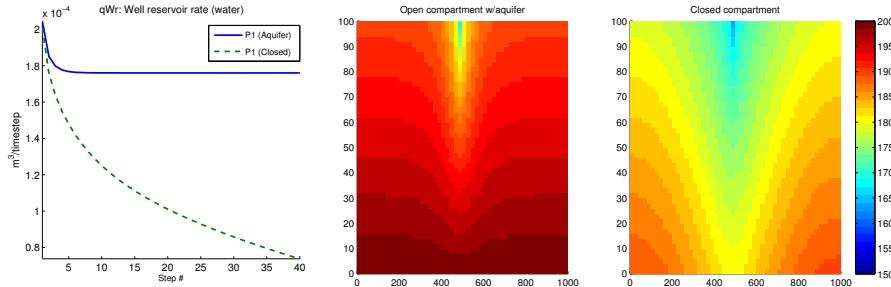


Fig. 12.6. Drainage from a single reservoir compartment that is either sealed or supported by an infinite aquifer from below.

We then create one simulation schedule for each setup, one with well and constant pressure on the south boundary, and the other with well and no-flow conditions on all boundaries:

```
schedule1 = simpleSchedule(diff(linspace(0,5*day,41)), 'bc', bc, 'W', w);
schedule2 = simpleSchedule(diff(linspace(0,5*day,41)), 'W', w);
```

The pristine reservoir will be in vertical equilibrium, which we compute as a steady-state solution of the flow equation subject to boundary conditions only

```
state.pressure = ones(G.cells.num,1)*pR;
state.wellSol = initWellSolAD([], wModel, state);
nonlinear = NonLinearSolver;
state = nonlinear.solveTimestep(state, 10000*day, wModel, 'bc', bc);
```

We can then compute each flow solution as follows:

```
[wellSols1, states1] = simulateScheduleAD(state, wModel, schedule1);
```

Figure 12.6 shows the solution of the two problems. Once the well is turned on, the draw-down at the wellbore will create an elongated pressure pulse that moves fast downward and expands slowly outward. The open compartment will quickly reach a steady-state in which the flow rate into the well equals the influx across the bottom boundary. In the closed case, the pressure inside the compartment will gradually be depleted towards the perforation pressure, which in turn will cause the production rate to decline towards zero.

12.4.2 An undersaturated sector model

The next example² continues in the same vein as the previous: we consider a $1000 \times 1000 \times 100$ m³ rectangular sector model filled with undersaturated oil, and compare and contrast the difference in production when the boundaries are closed or held at constant pressure. The phase behavior of the oil and

² Complete code: `blackoilSectorModelExample.m` in `ad-blackoil`

dissolved gas is assumed to follow that of the SPE 1 benchmark, but unlike in the original setup, the water is assumed to be mobile, following a simple quadratic relative permeability.

```
[~, ~, fluid, deck] = setupSPE1(); fluid.krW = @s(s.^2);
```

The permeability is homogeneous and isotropic, and the fluid and pressure is for simplicity assumed to be uniformly distributed inside the reservoir:

```
[s0, p0] = deal([0.2, 0.8, 0], 300*barsa);
state0 = initResSol(G, p0, s0);
state0.rs = repmat(200, G.cells.num, 1);
model = ThreePhaseBlackOilModel(G, rock, fluid, 'disgas', true);
```

The reservoir will be produced from a single vertical well, placed in the center of the reservoir and operating at a fixed oil rate. However, we also define a lower limit on the bottom-hole pressure

```
ij = ceil(G.cartDims./2);
W = verticalWell([], G, rock, ij(1), ij(2), [], ...
    'val', -0.25*sum(model.operators.pv)/T, ...
    'type', 'orat', 'comp_i', [1, 1, 1]/3, ...
    'sign', -1, 'name', 'Producer');
```

Also the AD-OO framework assumes no-flow boundary conditions by default, and hence no boundary conditions must be specified to simulate closed boundaries. For the other case, we make gross simplification and assume that the pressure and fluid composition are held at their initial values to model additional pressure support, e.g., from a gas cap or an aquifer with highly mobile water. Although such boundary conditions have been used often in the past, there is in reality no physical mechanism that would be able to maintain constant boundary conditions in a reservoir, so this is a gross simplification that should not be applied when modelling real cases.

```
bc = [] ; sides = {'xmin', 'xmax', 'ymin', 'ymax'};
for side = 1:numel(sides)
    bc = pside(bc, G, sides{side}, p0, 'sat', s0);
end
```

We also need to specify the R_s value on all boundaries. For this purpose, MRST uses a dissolution matrix, which can either be specified per cell interface, or for all interfaces. Here, the fluid composition is the same on the whole boundary, so we only specify a single matrix

```
bc.dissolution = [1, 0, 0; ... % Water fractions in phases
                  0, 1, 0; ... % Oil fractions in phases
                  0, 200, 1]; % Gas fractions in phases
```

To define specific dissolution factors on n individual cell faces, we would have to set up an $n \times 3 \times 3$ array.

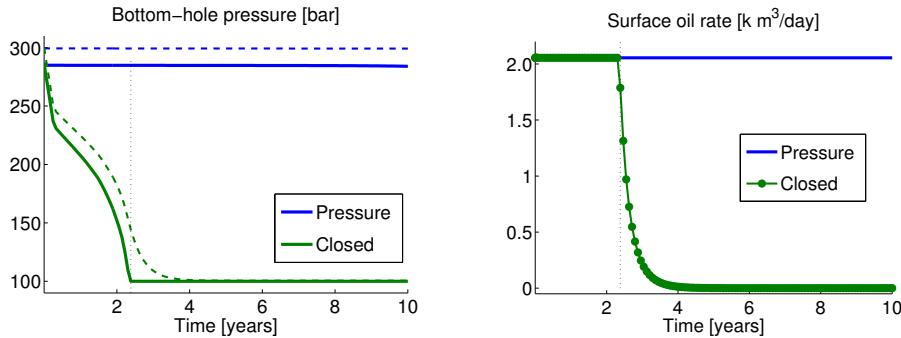


Fig. 12.7. Bottom-hole pressure and surface oil rate for the undersaturated sector model; dashed lines show field-average pressure. With closed boundaries, the well control switches from oil surface rate to bottom-hole pressure after 870 days (shown as dotted lines).

As in the previous case, turning on the well will induce a fast pressure transient that lasts a few days. We will not try to resolve this, but to improve the initial stability of the simulation, we gradually ramp up the time step, as discussed earlier in the book:

```
dt = rampupTimesteps(T, 30*day);
```

Specifically, the routine splits the simulation horizon T into a number of time steps of length 30 days (the last step may be shorter if T is not a multiple of the specified time step), and then uses a geometric sequence $1./2.^{[n n:-1:1]}$ to further subdivide the first time step (default value of n is 8). We can now set up and simulate the two schedules in the same way as above

```
schedule = simpleSchedule(dt, 'W', W, 'bc', bc);
[ws, states] = simulateScheduleAD(state0, model, schedule);
```

Figure 12.7 shows bottom-hole pressure and surface oil rate for the two simulations, i.e., the quantities used to control the production well. After the initial transient, the case with constant pressure boundary will settle at a steady state with constant bottom-hole pressure and oil rate maintained at its target value. With closed boundaries, there is no mechanism to maintain the reservoir pressure and hence the bottom-hole pressure (bhp) will start to gradually decay. After 870 days, the bhp drops below its lower limit of 100 bar, and the well switches control from oil rate to bhp. As production continues, the reservoir pressure will gradually be depleted until the point where there is not sufficient pressure difference between the reservoir and the well to drive any flow into the wellbore; this happens after approximately four and a half years. Grossly speaking, qualitative behavior of the pressure is similar to what we observed in the previous single-phase example.

Figure 12.8 reports cumulative production of oil, gas, and water at surface conditions, as well as instantaneous production rates of the oleic, gaseous, and aqueous phases at reservoir conditions. As the reservoir pressure is gradually

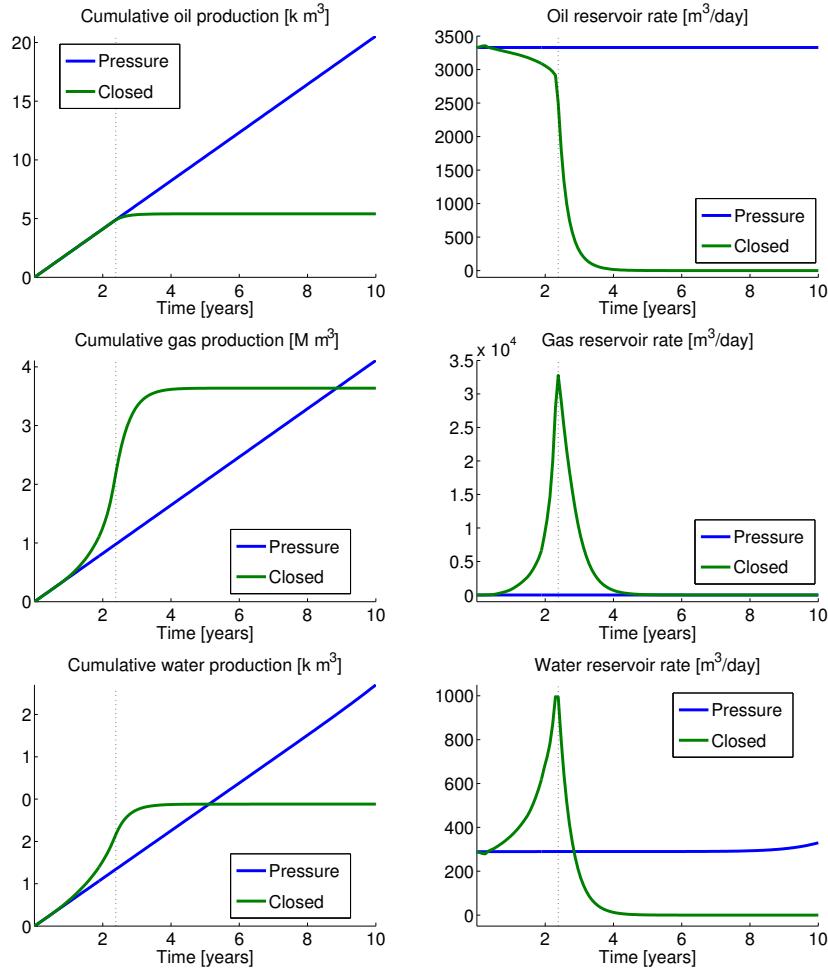


Fig. 12.8. Production profiles for the undersaturated sector model. The left column shows cumulative production of the oil, gas, and water components at surface conditions, whereas the right column shows instantaneous phase rates at reservoir conditions.

lowered, some cells drop below the bubblepoint and free gas will be liberated, starting near the wellbore, where the pressure drop is largest, and gradually moving outward as shown in Figure 12.9.

Oil is only produced from the oleic phase. When free gas is liberated, the fraction of oil in the oleic phase will increase, and the oil reservoir rate will decay slightly, because the reservoir volume of the oleic phase required to produce a constant surface rate of oil decreases. The liberated gas is very mobile and will cause an accelerating pressure drop as more gas starts flowing

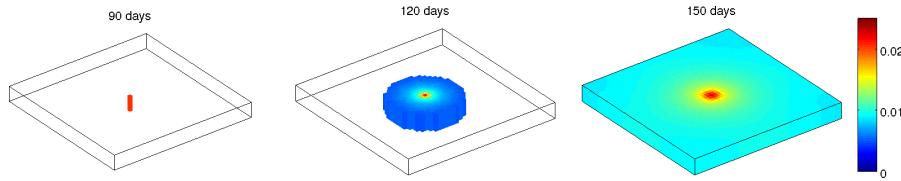


Fig. 12.9. Liberation of free gas as the drawdown from the well causes the pressure in the undersaturated oil to drop below the bubblepoint. (The plot shows S_g .)

into the wellbore. Gas production at the surface thus increases dramatically, because gas is now produced both from the oleic and the gaseous phase. Gas can sometimes be sold and contribute to generate revenue, but the operator may also have to dispose of it by reinjecting it into the reservoir to maintain reservoir pressure. Unlike in the original SPE 1 case, water is mobile. As gas is liberated the mobility of oil is reduced and we see a significant increase also in water production. Topside facilities will often have limited capacity for handling produced water, and the whole production may reduced or shut down if the water rate exceeds the available capacity. Here, it would have been advantageous if the well was instrumented with autonomous inflow devices that would choke back water and gas and only let oil flow through. We will return to this in the next example.

As the well switches to bhp control, all rates fall rapidly. For oil, in particular, the additional recovery after the well switches is very limited: at 870 days, the cumulative oil production has reach 93% of what is predicted to be the total production after 10 years. The liberated gas (and the water) will continue to flow longer, but also here the production ceases more or less after four and a half year.

This idealized example is merely intended to demonstrate use of the AD-OO framework and illustrate basic flow mechanisms and makes no claim of being representative of real hydrocarbon reservoirs.

COMPUTER EXERCISES:

80. Try to simulate the initial pressure transient, so check how long it lasts and whether it behaves as in the previous example.
81. The simulation above did not include any consideration of production relative to the amount of fluids initially in place. Try to include this to determine the recovery factor for the closed case. Can you also use this to say something about the influx across the boundaries in the case with pressure boundaries.
82. With constant pressure boundaries, the water rate starts to increase after approximately seven years. Can you explain this behavior?
83. Gravity was not included in this example. Can you set up multilayer simulations to investigate if gravity has a significant effect on the result?

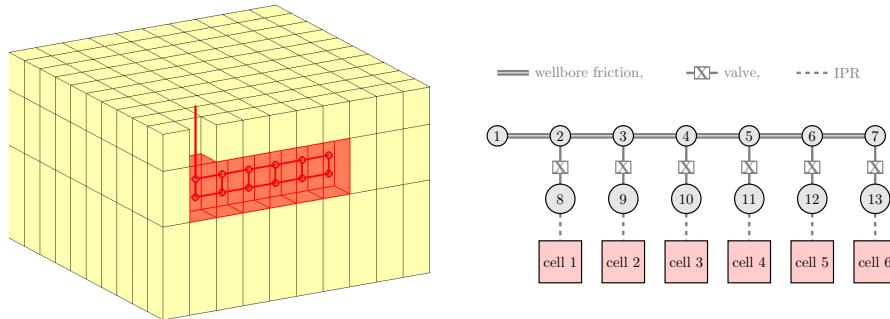


Fig. 12.10. Multisegment well for the modified SPE 1 case. Nodes 1 to 7 represent the wellbore, whereas nodes 8 to 13 represent the void outside of the valve that separates the wellbore from the sandface.

12.4.3 SPE 1 with a multisegment well

This example³ demonstrates the use of multisegment wells in MRST. The default well model in MRST assumes that the flow in the well takes place on a very short time-scale compared to the reservoir and thus can be accurately modelled as being instantaneous following the linear inflow performance relationship discussed in Section 11.6.1. Such models may not be appropriate, e.g., for long wellbores that have significant pressure drop due to friction, or for wells having valves or more sophisticated inflow control devices. The multisegment well class in MRST supports transient flow inside the well itself, so that the effects of friction, acceleration, and nonlinear pressure drops across valves and other flow constrictions can be included in the well model; see Section 11.6.2.

To illustrate, we revisit the SPE 1 case from Section 11.7.1 and replace the single-cell, vertical producer by a horizontal well completed in six cells with valves between the production tubing and each sandface connection. We consider two models, a simple model assuming instantaneous flow with a standard inflow relationship and hydrostatic pressure along the wellbore, and a more advanced model that accounts for pressure drop across the valves and frictional pressure drop along the horizontal wellbore, see Figure 12.10.

We start by loading the model and the simulation schedule from an ECLIPSE input deck

```
[G, rock, fluid, deck, state] = setupSPE1();
[nx, ny] = deal(G.cartDims(1),G.cartDims(2));

model = selectModelFromDeck(G, rock, fluid, deck);
model.extraStateOutput = true;

schedule = convertDeckScheduleToMRST(model, deck);
```

³ Complete source code: `multisegmentWellExample.m` in `ad-blackoil`

To specify the wells, we first initialize the production well as a standard well structure completed in the six given grid cells:

```
c = nx*ny + (2:7)';
prod0 = addWell([], G, rock, c, 'name', 'prod', ...
    'refDepth', G.cells.centroids(1,3), ...
    'type', 'rate', 'val', -8e5/day);
```

We then define the N_1, N_2 mappings for the twelve segments, as well as the mapping from network nodes to reservoir cells represented in the well structure

```
topo = [1 2 3 4 5 6 2 3 4 5 6 7
        2 3 4 5 6 7 8 9 10 11 12 13]';
% | tubing | valves |
cell2node = spares((8:13)', (1:6)', 1, 13, 6);
```

The latter sparse matrix will have dimension equal number of nodes times number of reservoir cells perforated by the well. We also need to specify the segment lengths and diameters and the node depths and volumes

```
lengths = [300*ones(6,1); nan(6,1)];
diam = [.1*ones(6,1); nan(6,1)];
depths = G.cells.centroids(c([1 1:end 1:end]), 3);
vols = ones(13,1);
```

With this, we have all the information we need to convert the data structure of the simple well into a corresponding data structure describing a multisegment network model

```
prodMS = convert2MSWell(prodS, 'cell2node', cell2node, 'topo', topo, ...
    'G', G, 'vol', vols, 'nodeDepth', depths, ...
    'segLength', lengths, 'segDiam', diam);
```

Finally, we need to set up the flow model for each of the segments. The first six segments will be modelled using the wellbore-fiction model (11.29)–(11.31) with roughness $e = 10^{-4}$ m and discharge coefficient $C_v = 0.7$. The next six segments represent valves, across which the pressure loss is modelled by use of (11.29) with mixture mass flux. The valves have thirty openings, each with a nozzle diameter of 25 mm. We then set up the flow model as function of velocity, density, and viscosity

```
[wbix, vix] = deal(1:6, 7:12);
[roughness, nozzleD, discharge, nValves] = deal(1e-4, .0025, .7, 30);

prodMS.segments.flowModel = @(v, rho, mu)...
    [wellBoreFriction(v(wbix), rho(wbix), mu(wbix), ...
        prodMS.segments.diam(wbix), ...
        prodMS.segments.length(wbix), roughness, 'massRate'); ...
    nozzleValve(v(vix)/nValves, rho(vix), nozzleD, discharge, 'massRate')];
```

The injector is a simple well modelled using the standard inflow performance relationship

```
inj = addWell([], G, rock, 100, 'name', 'inj', 'type', 'rate',...
    'Comp_i', [0 0 1], 'val', 2.5e6/day,'refDepth', G.cells.centroids(1,3));
```

We will now run two different schedules: the first is a baseline simulation in which the producer is treated as a simple well with instantaneous flow and one node per contact between the well and the reservoir

```
schedule.control.W = [inj; prodS];
[wellSolsSimple, statesSimple] = simulateScheduleAD(state, model, schedule);
```

For the second simulation, we first need to combine the simple injector and multisegment producer into a facility model.

```
W = combineMSwithRegularWells(inj, prodMS);
schedule.control.W = W;
model.FacilityModel = model.FacilityModel.setupWells(W);
```

This is normally done automatically by the simulator, but we do it explicitly on the outside to view the output classes. These classes are practical if per-well adjustments to e.g., tolerances are desired. The classes and the well structures read

<pre>SimpleWell with properties:</pre> <pre> W: [1x1 struct] allowCrossflow: 1 allowSignChange: 0 allowControlSwitching: 1 dpMaxRel: Inf dpMaxAbs: Inf dsMaxAbs: 0.2000 VFPTable: [] operators: [] nonlinearTolerance: 1.0000e-06 G: [] verbose: 0 stepFunctionIsLinear: 0 </pre> <pre>W is structure with fields: cells: 100 type: 'rate' val: 28.9352 r: 0.1000 : cell2node: [] connDZ: [] isMS: 0 nodes: [] segments: [] </pre>	<pre>MultisegmentWell with properties:</pre> <pre> signChangeChop: 0 W: [1x1 struct] allowCrossflow: 1 allowSignChange: 0 allowControlSwitching: 1 dpMaxRel: Inf dpMaxAbs: Inf dsMaxAbs: 0.2000 VFPTable: [] operators: [1x1 struct] nonlinearTolerance: 1.0000e-06 G: [] verbose: 0 stepFunctionIsLinear: 0 </pre> <pre>W is structure with fields: cells: [6x1 double] type: 'bhp' val: 25000000 r: 0.1000 : cell2node: [13x6 double] connDZ: [6x1 double] isMS: 1 nodes: [1x1 struct] segments: [1x1 struct] </pre>
--	--

The main difference in the two classes is that the simple model does not have any operators defined. Looking at the structure `W`, the simple model does not have any representation of cell to node mappings and structures representing the nodes and segments. However, what MRST uses to distinguish simple and

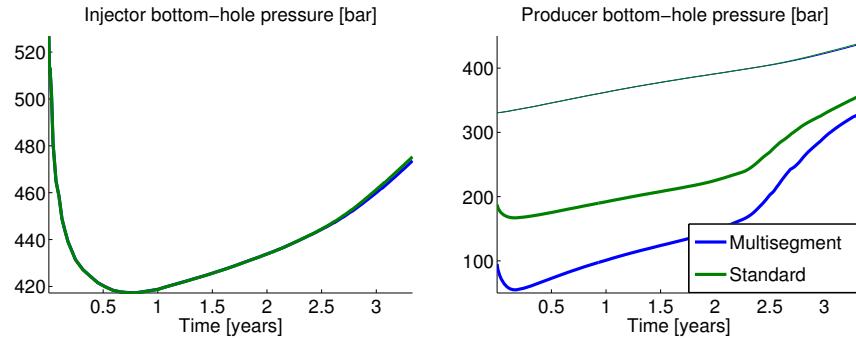


Fig. 12.11. Comparison of bottom-hole pressures in the injector (left) and the producer (right) predicted for the modified SPE 1 model with a simple well model and a well model accounting for pressure drop across valves and caused by friction along the horizontal wellbore. Thin lines in the right plot show average-field pressure.

multisegment wells is the flag `isMS`, which is set to false for the injector and true for the producer. For completeness, let us also look at the data structures representing nodes and segments

segments: length: [12x1 double] roughness: [12x1 double] diam: [12x1 double] flowModel: [function_handle] topo: [12x2 double] deltaDistance: [12x1 double]	nodes: depth: [13x1 double] vol: [13x1 double] dist: [13x1 double]
---	--

The crucial element here is the function handle `flowModel` in the segments, which is used to assemble the well equations.

Figure 12.11 shows bottom-hole pressure in the producer and injector as well as field-average pressure predicted by the two different well models. Resistance to flow when the flow is constricted through the nozzles of the valves and by friction along the tubing causes a significantly higher pressure drop from the sandface to the datum point in the wellbore. Hence, the bhp will be significantly lower for the multisegment model compared with the simple well model to maintain the same surface production rate⁴. Looking at the injector bhp, we see that a relatively high pressure is needed initially to force the first gas into the reservoir. As soon as the first gas has entered the reservoir, the total mobility in the near-well region increases and the injection pressure decays. With some gas accumulated, the total flow resistance of the pore space from injector to producer will decrease and hence the bottom-hole pressure starts to rise in the producer after approximately 60 days. On the other hand,

⁴ Notice that this comparison is somewhat exaggerated. In reality, one would likely have adjusted either the well index or the skin factor in the simple well model to account for some of this pressure loss. On the other hand, such representative values can only be found after one has observed some production history.

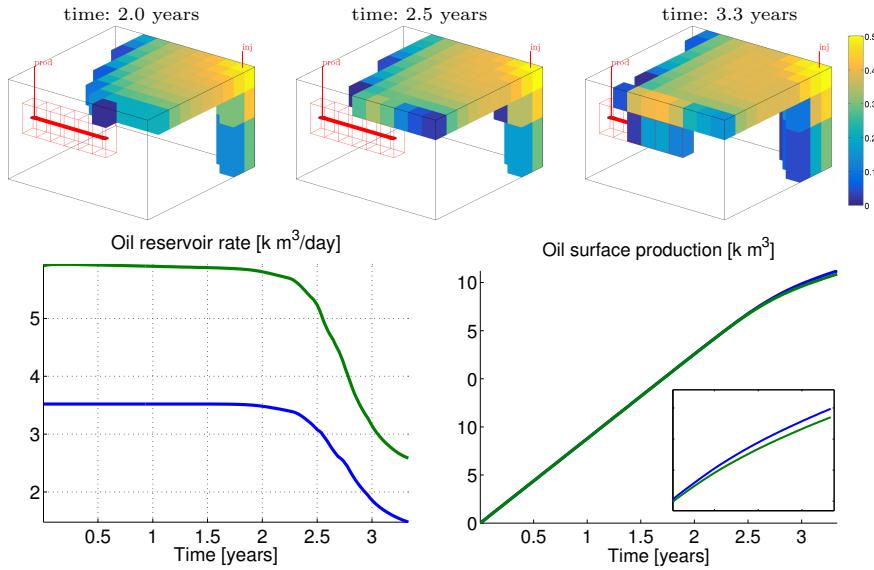


Fig. 12.12. Displacement front engulfing the horizontal producer causes decay in oil production. Because of different pressure in the wellbore, the volumetric reservoir rates will be largely different in the two models, even though the surface rates are almost identical.

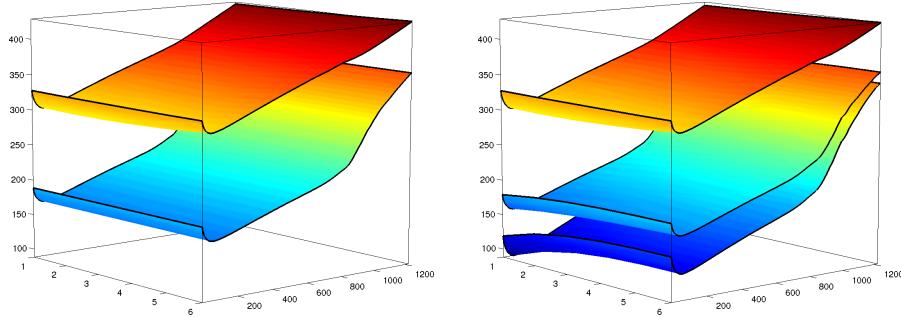


Fig. 12.13. Pressure drawdown from the perforated cells to the wellbore as function of time for the simple well model (left) and the multisegment well model (right). In the latter plot, the middle surface represent pressure outside of the valves.

the injected gas is not able to displace a similar amount of reservoir fluids from the reservoir because of lower density and higher compressional effects, and hence the reservoir pressure will gradually increase. This effect is observed in both models.

After approximately twenty months, the first gas has reached a neighboring cell of the producer and will start to engulf the cells above and gradually be drawn down into the perforated cells, see Figure 12.12. This engulfing process

causes a significant decay in the oil rate and a steeper incline in producer bhp. The rate decay is slightly delayed in the multisegment model, but the effect on the total oil production is a mere 1.3% increase.

Figure 12.13 shows drawdown for both well models. Because the simple well model assumes instantaneous hydrostatic fluid distribution and the producer is completely horizontal, the pressure will be identical in all the six points along the wellbore that represent individual connections with the reservoir. For the multisegment well model, we observe a significantly larger pressure drawdown near the heel of the well, which also lies furthest away from the injector and the advancing gas front. Both models use the same inflow-performance relationship and hence the drawdown outside of the valve in the multisegment model will on average be approximately the same as for the simple well model when both wells are run on rate control

COMPUTER EXERCISES:

80. Our setup had no lower limit on the bottom-hole pressure. What happens if bhp in the producer is not allowed to drop below 100 bar or 150 bar?
81. Rerun the two simulations with producer controlled by a bottom-hole pressure of 250 bar. Can you explain the differences in the producer and in the bhp of the injector?

12.4.4 The SPE 9 benchmark case

In our last example⁵, we set up a more realistic simulation case, show how to use CPR preconditioning together with an algebraic multigrid solver, and compare our simulation results against ECLIPSE to verify that the AD-OO black-oil simulator is correct. To this end, we consider the model from the SPE 9 comparative solution project, which was posed in the mid 1990's to compare contemporary black-oil simulators [131]. The benchmark consists of a water injection problem in a highly heterogeneous reservoir described by a $24 \times 25 \times 15$ regular grid, having a 10 degree dipping-angle in the x -direction. By current standards, the model is quite small, but contains several salient features that must be properly handled by any black-oil simulator aiming to solve real problems. On one hand, the fluid and PVT model have certain peculiarities; we have already discussed these models in detail in Sections 11.2 and 11.4, respectively. In particular, the discontinuity in the water-oil capillary pressure curve is expected to challenge the Newton solver when saturations change significantly. Other challenges include strong heterogeneity, transition from undersaturated to saturated state, abrupt changes in prescribed production rates, and dynamic switching between rate and bhp control for the producers. Altogether, this makes the case challenging to simulate.

⁵ Complete source code: `blackoilTutorialSPE9.m` in `ad-blackoil`

Petrophysical model and initial conditions

You may recall that we have already studied the fluid and PVT model respectively. In the following, we will show how to set up the full simulation case and discuss the dynamic of the problem. The basic setup of the grid, petrophysical parameters, fluid model, and initial state is more or less identical to that of the SPE 1 model discussed in Section 11.7.1. For convenience, this is implemented in a dedicated function:

```
[G, rock, fluid, deck, state0] = setupSPE9();
model    = selectModelFromDeck(G, rock, fluid, deck);
schedule = convertDeckScheduleToMRST(model, deck);
```

The reservoir rock is highly heterogeneous, albeit not as bad as the SPE 10 benchmark. The model consists of fourteen unique layers of varying thickness. The permeability is isotropic and follows a lognormal distribution with values spanning more than six orders of magnitude, see Figure 12.14. The porosity field has no lateral variation within each layer. One of the the high-permeable layers is represented by two cells in the vertical direction, whereas two of the other layers have the same porosity, altogether giving twelve unique porosities.

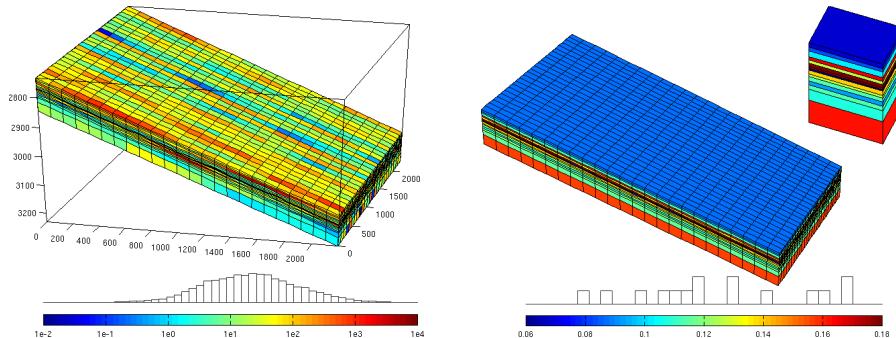


Fig. 12.14. Petrophysical data for the SPE 9 model: the left plot shows absolute permeability in units md and the right plot shows porosity.

Initially, the reservoir does not contain any free gas, but a significant amount of gas is dissolved into the oleic phase. The dissolved amount r_s is constant throughout the whole model, except for three individual cells. Some of the dissolved gas may be liberated as free gas if the reservoir pressure drops below the bubblepoint during production. As we know, the solution gas-oil ratio R_s varies with pressure, which in turn increases with depth. At the top of the reservoir, the oleic phase is fully saturated ($r_s = R_s$), whereas the degree of saturation is only 89% at the bottom of the reservoir ($r_s = 0.89R_s$). We can therefore expect that some free gas will form at the top of the reservoir once the wells start producing.

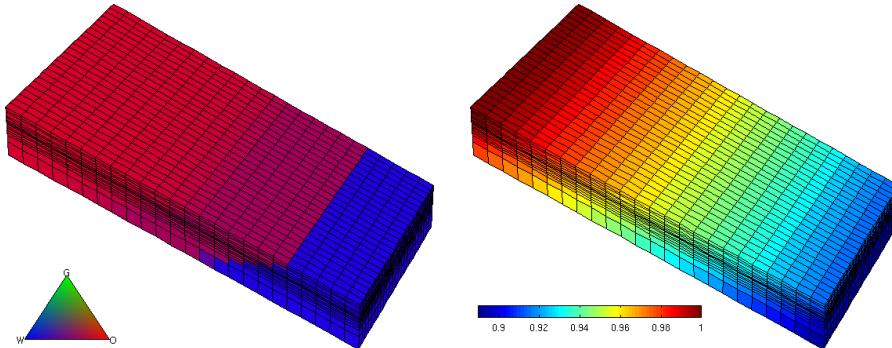


Fig. 12.15. Initial fluid distribution inside the reservoir. The left figure shows the saturation, which lies on the two-phase O–W axis in the ternary diagram, whereas the right plot shows the degree of saturation defined as $r_s/R_s(p)$.

Wells and simulation schedule

The reservoir is produced from 25 vertical wells that initially operate at a maximum rate of 1500 STBO/D (standard barrels of oil per day). Pressure is supported by a single water injector, operating at a maximum rate of 5000 STBW/D (standard barrels of water per day) and a maximum bottom-hole pressure of 4000 psi at reference depth. Between days 300 and 360, the production rate is lowered to 100 STBO/D, and then raised up again to its initial value until the end of simulation at 900 days. Throughout the simulation, most of the wells switch from rate control to pressure control.

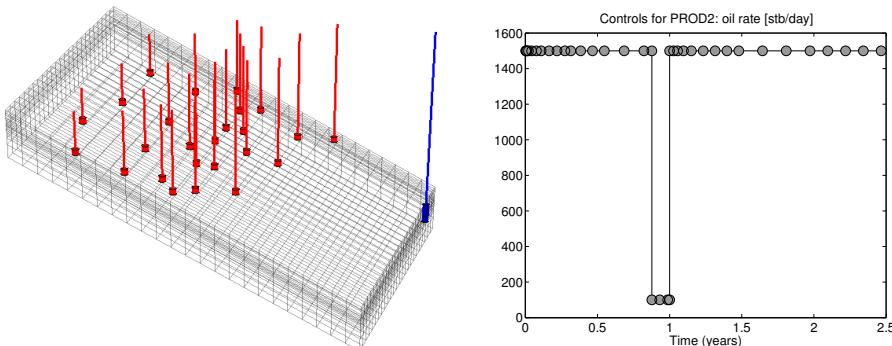


Fig. 12.16. Well placement in the reservoir (left) and simulation schedule with rate targets for the producers (right).

The simulation schedule consists of three control periods: day 0 to 300, day 300 to 360, and day 360 to 900, with a ramp-up of time steps at the beginning of the first and third period. All 26 wells are present during the

entire simulation. The injector is injecting a constant water rate, while the producers are set to produce a target oil rate, letting bottom-hole pressures and gas/water production vary. In addition to the rate targets, the producers are limited by a lower bhp value of 1000 psi, which could e.g., signify the lowest pressure we can have in the well and still be able to lift reservoir fluids to the surface. The prescribed limits can be found in the `lims` field of each well. For producer number two, this would be `schedule.control(1).W(22).lims`

```
Well limits for PROD2:
orat: -0.0028
wrat: -Inf
grat: -Inf
lrat: -Inf
bhp: 6.8948e+06
```

By convention, the oil rate has negative sign for a producer. The pressure control will be activated if the wellbore pressure falls below the lower limit

Customizing linear solver and time-step control

The model has 9 000 active cells, which for a three-phase problem corresponds to a linear system with 27 000 cell unknowns; the total number of unknowns is slightly higher because of the 26 wells. Models of this size can be solved with the standard direct solvers in MATLAB on most computers, but run-times can be reduced if we instead use a CPR-preconditioner. To solve the elliptic pressure subsystem, we will use the algebraic multigrid solver AGMG, with MATLAB's standard direct solver as potential backup, if AGMG is not installed on the computer:

```
try
    mrstModule add agmg
    pressureSolver = AGMGSolverAD('tolerance', 1e-4);
catch
    pressureSolver = BackslashSolverAD();
end
linsolve = CPRSolverAD('ellipticSolver', pressureSolver, ...
    'relativeTolerance', 1e-3);
```

We also adjust the default time-step control slightly to ensure stable simulation by tightening the restriction on relative changes pressure and absolute changes in saturation

```
model.dpMaxRel = .1;
model.dsMaxAbs = .1;
```

Beyond this, no modifications are made to the standard configuration of the simulator, which uses robust defaults from ECLIPSE.

Running the simulation

The simulation can now be started by use of the following statements

```
model.verbose = true;
[wellsols, states, reports] =...
simulateScheduleAD(state0, model, schedule,'LinearSolver', linsolve);
```

We give the schedule with well controls and control time steps. The simulator may use other time steps internally, but it will always return values at the specified control steps. Turning on verbose mode ensures that we get an extensive report of the nonlinear iteration process for each time step:

```
Solving timestep 01/35: -> 1 Day
Well INJE1: Control mode changed from rate to bhp.
=====
| It # | CNV_W | CNV_O | CNV_G | MB_W | MB_O | MB_G | waterWell| oilWells | gasWells | closure |
=====
| 1 | 2.32e-02 | 1.10e+00 | 5.00e-03 | 2.00e-04 | 1.58e-06 | 6.81e-04 | 5.00e-02 | 1.73e-01 | *0.00e+00 |
| 2 | 1.14e-01 | 7.25e-02 | 5.47e-02 | 5.17e-06 | 1.48e-04 | 8.86e-04 | 4.40e-03 | 2.20e-03 | 1.47e+06 | Well PROD26: Control mode changed from orat to bhp.
| 3 | *6.37e-04 | 2.38e-02 | 6.68e-02 | 2.24e-04 | 1.13e-03 | *6.04e-07 | 4.58e-05 | 6.02e-02 | *1.73e-18 | |
| 4 | 1.89e-03 | 8.12e-03 | 2.55e-02 | 7.23e-06 | 4.74e-05 | 3.56e-04 | *5.09e-06 | 1.85e-04 | 5.16e-02 | 5.67e+05 |
| 5 | *6.04e-04 | 1.69e-03 | *6.50e-04 | 1.32e-06 | 3.47e-06 | 1.90e-06 | *5.17e-07 | 1.92e-05 | 4.85e-03 | *8.67e-19 |
| 6 | *6.31e-04 | 1.46e-03 | *6.72e-04 | 1.16e-06 | 4.47e-06 | 1.74e-06 | *2.28e-10 | 1.56e-05 | 3.87e-03 | *0.00e+00 |
| 7 | *9.16e-05 | *3.79e-04 | *2.20e-04 | 7.16e-04 | 3.82e-07 | *9.41e-09 | *3.58e-11 | *2.41e-07 | 6.84e-05 | *0.00e+00 |
| 8 | *7.96e-05 | *2.45e-04 | *1.08e-04 | *5.50e-08 | 2.40e-07 | *7.63e-08 | *2.87e-14 | *3.82e-11 | *1.10e-08 | *0.00e+00 |
| 9 | *1.37e-05 | *5.70e-05 | *3.43e-05 | *2.15e-09 | *1.20e-09 | *1.22e-09 | *1.79e-14 | *7.86e-11 | *2.40e-08 | *0.00e+00 |
=====

Solving timestep 02/35: 1 Day -> 2 Days
```

Here, we recognize the mass-balance (MB) errors and the maximum normalized residuals (CNV) discussed on page 463 for water, oil, and gas. The next three columns give residuals for the inflow-performance relationship defined per perforations, whereas the last column gives residual for the control equations defined per well. A star means that the residual is below the tolerance. For the first time step, the last measure to converge was the mass-balance error for oil was the last one to drop below the prescribed value of 10^{-9} . Nine iterations is quite high number and indicates that there are significant nonlinearity in the residual equation, which here primarily is caused by initial transient effects. The next seven control steps all converge within 6 iterations. Between days 80 and 100, liberated gas reaches the water zone and the next two steps now require 7 and 12 iterations. Large jumps in the iteration count can typically be traced back to interaction of different displacement fronts, abrupt phase transitions, and/or steep/discontinuous changes in the well controls or other parameters affecting the global drive mechanisms.

Switching of well controls

Figure 12.17 shows which control is active during each of the 35 control steps. Initially, all wells are set to be rate controlled. The targeted injection rate of 5000 STBW/day is much higher than what can be achieved with reasonable injection pressures, and hence the injector switch immediately from rate to bhp control during the first time step and stays there throughout the whole simulation. Likewise, producer PROD26 and producers PROD21 and PROD23 switch from rate to bhp control during the first and second control step, respectively. This switching is reported to screen during simulation, and in the

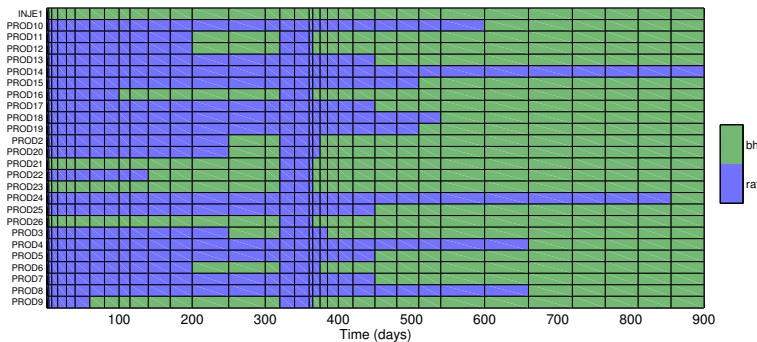


Fig. 12.17. Well controls active during the simulation of the SPE 9 benchmark.

report on the facing page, we see that the injector switches during the first iteration and producer PROD26 after the second iteration. The overall picture is that producers are mostly rate controlled in the beginning and mostly controlled by bhp at the end; the only exception is PROD14, which runs rate control the whole period. This general switch from rate to bhp control is a result of the average field pressure dropping during the simulation as fluids are removed from the reservoir. We also notice the middle period during which rates are reduced by a factor ten. A much smaller drawdown is required to maintain this rate, and hence all producers are able to stay above the bhp limit.

Analysis of simulation results

This reservoir is an example of a combined drive, in which two different mechanisms contribute to push hydrocarbons towards the producers. The first is injection of water below the initial oil–water contact, which will contribute to maintain the reservoir pressure and push water into the oil zone to displace oil. Ideally, this would have been a piston-like displacement that would gradually push the oil–water contact upward, but because of heterogeneity and nonlinear fluid mobilities, water will displace oil as a collection of ‘fingers’ extending from the oil–water contact and towards the producers, see Figure 12.18. Because the injector operates at fixed pressure, the injection rate will increase monotonically as the reservoir is gradually depleted

The second drive mechanism is liberation of free gas, which will start near the producers, where the pressure drawdown is largest. As pressure continues to decay, more gas will be liberated at the top of the formation where the pristine oil phase is closest to a saturated state. The liberated gas will form a gas cap that gradually grows downward and contributes to push more gas into the wells and reduce the production rates of oil, see the left column in Figure 12.19. Any produced and free gas must come from the initially dissolved gas in oil, since no gas has been injected. Figure 12.19 reports the amount of dissolved gas initially and after 900 days.

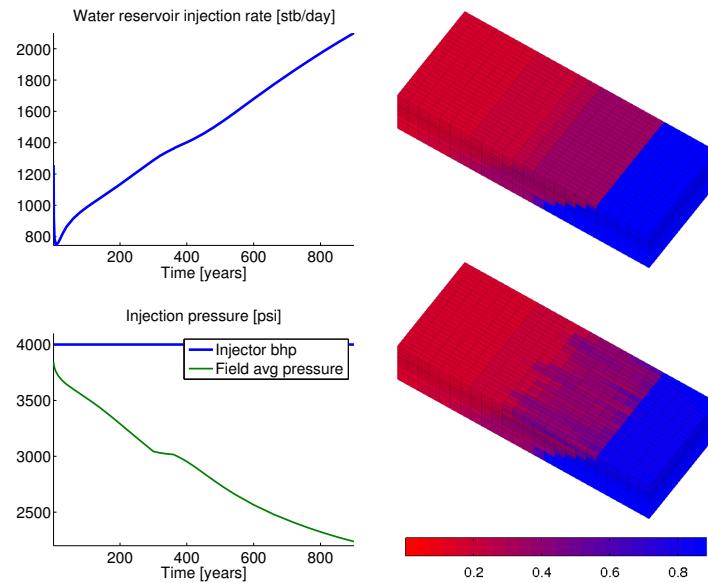


Fig. 12.18. Water drive contributing to enhanced production. The left column shows surface injection rate (top) and bottom-hole pressure compared with the average field pressure (bottom). The right column shows the bottom water drive, with initial water saturation at the top and water saturation after 900 days at the bottom..

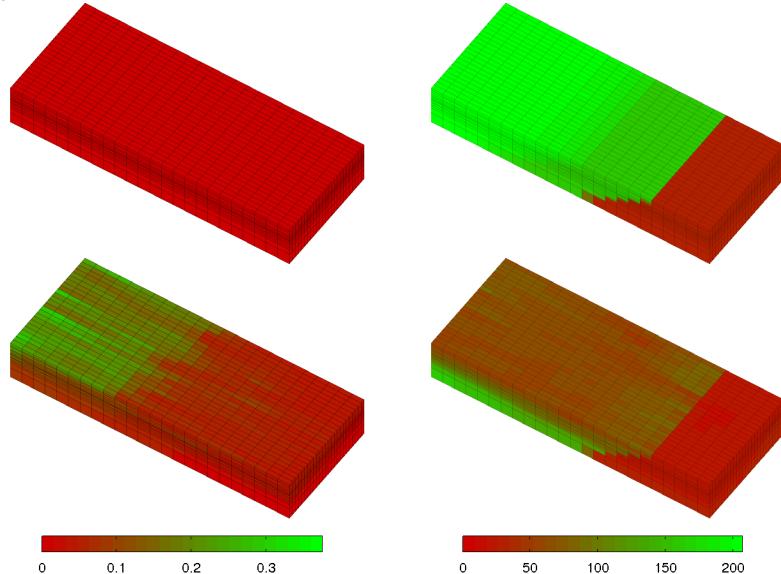


Fig. 12.19. Liberation of gas. The left column shows initial gas saturation at the top and gas saturation after 900 days at the bottom. The right column shows the amount of gas dissolved in the oleic phase.

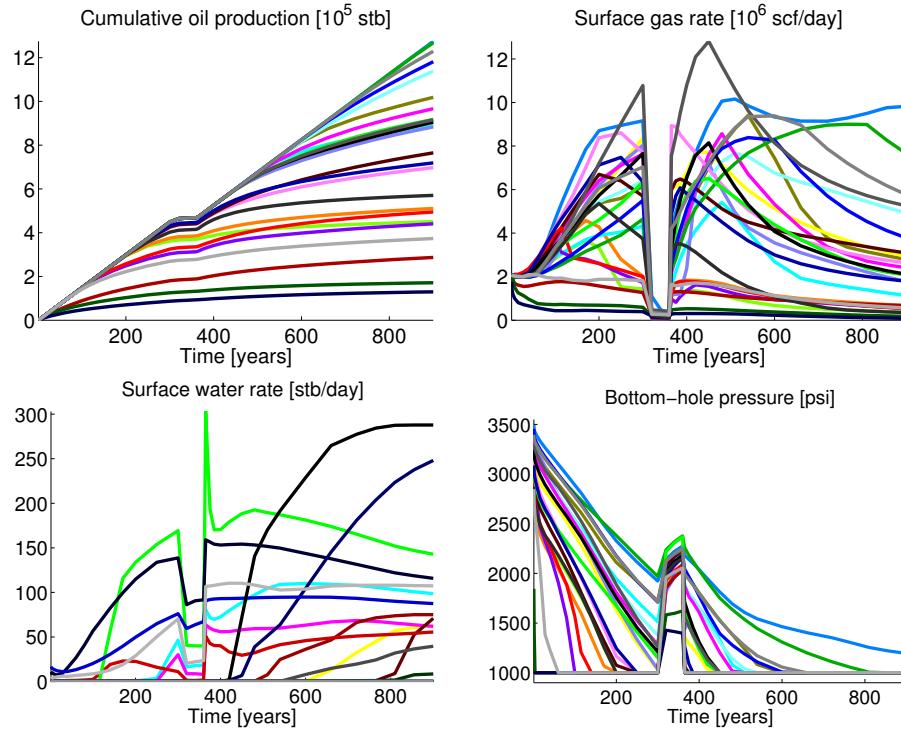


Fig. 12.20. Fluid production from the 25 producers.

Figure 12.20 shows production from the 25 wells. First of all, we note that there is significant spread in the recovery from the individual wells, with some wells that experience declining oil rates almost immediately and deviate from the upper production envelope that corresponds to the prescribed rate targets. These are wells at which free gas appears already during the first step. Since the free gas is more mobile and occupies a larger volume, it will effectively choke back the production of the oleic phase so much we see a *decay* in the total surface gas rates from these wells. Some of the other wells are able to maintain the prescribed oil rate for a longer period, and these wells experience increasing surface gas rates until the oil rate is reduced at 300 days. Other wells are of intermediate type, which first experience increasing and then decaying gas rates. Most wells only produce small amounts of water, and for several of those who produce significant amounts water, the breakthrough comes after the period with reduced oil rates has ended. The only exception in PROD17, which produces almost two orders of magnitude more water than the next producer (PROD17 is not shown in the lower-left plot in Figure 12.20). We note in passing that the plots of bottom-hole pressure confirms our previous discussion of switching well controls.

Verification against ECLIPSE

The default solvers in `ad-blackoil` have been implemented as to reproduce ECLIPSE as closely as possible. This does not necessarily mean that ECLIPSE always computes the *correct* result, but given the major role this simulator plays in the industry, it is very advantageous to be able to match results, so that this can be used as a baseline when investigating the performance of new models or computational methods. We have already shown that MRST matches ECLIPSE very well for the SPE 1 benchmark. To build further confidence in the simulator, we also show that MRST also matches ECLIPSE for the SPE 9 benchmark, given identical input. To this end, we will use the same functionality from the `deckreader` module for reading *restart* files as we used in Section 11.7.2:

```
addir = mrstPath('ad-blackoil');
compare = fullfile(addir, 'examples', 'spe9', 'compare');
smry = readEclipseSummaryUnFmt(fullfile(compare, 'SPE9'));
```

We first get the number of stored report steps, and the time at which these are computed

```
compd = 1:(size(smry.data, 2));
Tcomp = smry.get(':+:+:+:+', 'YEARS', compd);
```

Let us for instance compare the bottom-hole pressure of PROD13. The corresponding data can then be extracted as follows:

```
comp = convertFrom(smry.get('PROD13', 'WBHP', compd), psia);
mrst = getWellOutput(wellsols, 'bhp', 'PROD13');
```

and subsequently plotted using standard MATLAB commands as shown in Figure 12.21. The results computed by the two simulators are not identical. If you run the example, you will see that the ECLIPSE results contain 37 steps whereas the MRST output has 35. Unlike MRST, ECLIPSE includes the zeroth time step (i.e., the initial state), but more important uses one additional step during the ramp-up after the target oil rates have been increased back to their original value at 360 days. Nevertheless, the match between the two computed solutions appears to be almost perfect, and similar matches are also seen for the remaining 24 wells and other types of well responses.

Bao et al. [26] report similar good match for a set of polymer injection cases including non-Newtonian fluid rheology. We have run a number of cases in-house verifying that the AD-OO framework provides excellent match with commercial simulators. This includes challenging compositional cases as well as a full-fledged simulation model of the south wing of the Voador field in Brazil. Altogether, this shows that the software is a good starting point for case studies. However, when working with black-oil models, you should expect that you may stumble upon features that are not yet implemented in MRST, but this also the point where you can contribute by adding new features.

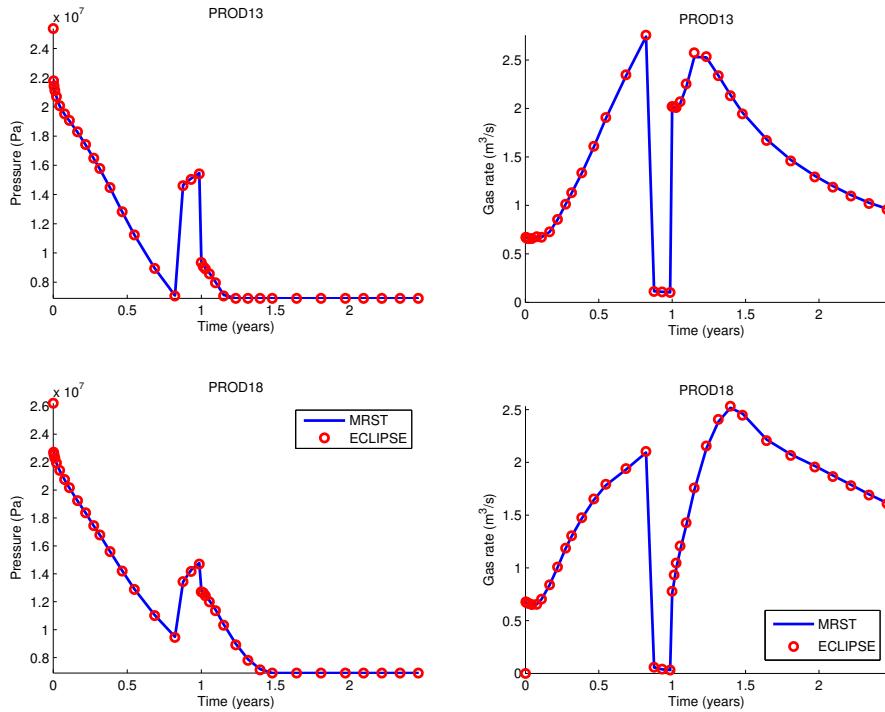


Fig. 12.21. Verification of the AD-OO black-oil solver from `ad-blackoil` against ECLIPSE. The left column shows bottom-hole pressures for two different producers, whereas the right column shows surface gas rates.

COMPUTER EXERCISES:

82. Set up an animation of water and gas to better understand how the aqueous and gaseous phases displace the oleic phase.
83. As pointed out above, well PROD17 produces orders of magnitude more water than other wells. How would shutting this well in from the start influence the overall field production.
84. Run the simulation using CPR with AGMG and MATLAB's direct solver as well as just using the direct solver for the whole system. Are there any noticeable differences in simulation times?
85. In some cases, produced gas cannot be sold but must be reinjected. Can you set up such a simulation based on the case discussed above? (Hint: Operations involving reinjection generally require complex topside logic to determine correct rates for the reinjected fluid. Such logic is not yet available in MRST. As a first approach, you can just assume that you have the gas readily available.)

12.5 Improving convergence and reducing runtime

I hope that this and the previous chapters have given you sufficient detail about MRST so that you can use it to perform interesting simulations. Although the software is primarily designed for flexibility and generality, the black-oil simulators implemented using the AD-OO framework offer such a comprehensive set of features and are so efficient that they can be used to simulate quite complicated models. As you explore other models than those discussed in the previous section, you will sooner or later run into convergence problems and/or cases for which the computational performance is not satisfactory. In this last section, I will briefly discuss how to address such problems and present a few recommendations that are not specific to MRST, but also apply to commercial simulators like ECLIPSE.

Tweaking control parameters for solvers

Like many other things in the AD-OO framework, the default values for the various parameters controlling the behavior of the `NonLinearSolver` class largely follow what is suggested for ECLIPSE. Default values in ECLIPSE have been developed based on many years of experience on a large variety of simulation models and will work well in most cases. Although you can observe improved performance of your simulation by tuning parameters in specific cases, it is generally *not recommended* that you change the default values unless you know what you are doing. In particular, it should seldom be necessary to *weaken* the convergence tolerances. When convergence problems occur in the nonlinear iterations, the culprit usually lurks in the time-step selection or the model itself, and it is too late to try to fix the problem by adjusting the convergence control. Contrary to what you may think, faster overall convergence can often be observed by *tightening* convergence controls rather than weakening them since this may prevent inaccuracies earlier in the simulation, which otherwise may accumulate from one time step to the next and cause severe errors later in the simulation.

As a general advice: the best way to improve convergence and computational performance is in most cases to choose a more robust linear solver and/or time-stepping scheme. In some cases, the cause of poor convergence and long runtimes may also lie in the model parameters, e.g., as discussed in Chapter 11.7.3.

Selection of control steps

How you set up your simulation schedule will largely determine the efficiency of your simulation. In many cases, the number of control steps and their individual lengths are determined by external factors. These can be requirements of monthly or quarterly reports for prediction runs, or the times of data points for history-matching runs. In other cases, there are no specific requirements and the schedule can be chosen more freely. The following guidelines may be useful to improve convergence and general computational performance:

- Make sure that the limit on the maximum time step is compatible with the specified control steps. A classical mistake is to set the maximum time step to be 30 days, and design a simulation schedule that reports results at the end of each month. As we all know, every second month has 31 days...
- For cases with complex flow physics (changes between undersaturated and saturated states, flow reversals, abrupt changes in drive mechanisms, etc), it may be advantageous to reduce long time steps since reductions in iteration count often will compensate for the increased number of time steps. As an example, using a gradual ramp-up of time steps initially and after major changes in well controls and other drive mechanisms will ensure that transients are better resolved, which in turn means that the simulation runs more smoothly
- In some cases, you may be asking for shorter control steps than you actually need. If, for instance, each control step consists of only one time step and/or the nonlinear solver converges very few iterations (say, one to three), it may be advantageous to increase the length of the control steps to give the time-step selector more freedom to choose optimal time steps.

For simulations based on ECLIPSE input decks, you should be particularly wary of the TUNING keyword, which is easy to misuse so that it interferes adversely with the standard mechanisms for time-step selection.

Part IV

Reservoir Engineering Workflows

13

Flow Diagnostics

Even for single-phase flow it is seldom sufficient to only study well responses and pressure distribution to understand the flow paths and communication patterns in a complex reservoir model. To gain a better qualitative picture of the flow taking place in the reservoir, you will typically want to know the answer to questions such as: From what region does a given producer drain? To what region does a given injector provide pressure support? Which injection and production wells are in communication? What part of the reservoir affects this communication? How much does each injector contribute to support the recovery from a given producer? Do any of the wells have back-flow? What is the sweep and displacement efficiency within a given drainage, sweep, or well-pair region? Which regions are likely to remain unswept? Likewise, you will typically also want to perform what-if and sensitivity analyzes to understand how different parameters in the reservoir model and their inherent sensitivity affect reservoir responses.

As we have seen earlier in the book, performing a single simulation of a full reservoir model containing a comprehensive description of geology, reservoir fluids, flow physics, well controls, and coupling to surface facilities is a computationally demanding task that may require hours or even days to complete. This means that your ability to study parameter variations is limited. This is particularly at odds with modern reservoir characterization techniques, in which hundreds of equiprobable realizations may be generated to quantify uncertainty in the characterization. In this chapter, we will therefore introduce a set of simple techniques referred to as *flow diagnostics* that can be used to develop basic understanding of how the fluid flow is affected by reservoir geology and how the flow patterns in the reservoir respond to engineering controls. In their basic setup, these techniques only rely on the solution of single-phase, incompressible flow problems as discussed in Chapters 4 and 5. (However, in parts of the presentation, we will use some simple concepts from multiphase flow that are introduced later in the book, but which are hopefully still understandable if you have not yet read about multiphase flow).

In general, flow diagnostics can be defined as *simple and controlled numerical flow experiments that are run to probe a reservoir model, establish connections and basic volume estimates, and quickly provide a qualitative picture of the flow patterns in the reservoir*. Flow diagnostics can also be used to compute quantitative information about the recovery process in settings somewhat simpler than what would be encountered in an actual field, and using these techniques, you can rapidly and iteratively perturb simulation input and evaluate the resulting changes in volumetric connections and communications to build an understanding of cause and effects in your model.

Ideas similar to what will be presented in the following have previously been used within streamline simulation [69] for ranking and upscaling [113, 22, 232], identifying reservoir compartmentalization [103], rate optimization [240, 208, 115], and flood surveillance [28]. Our presentation, however, is inspired by Shahvali et al. [230] and [184], who developed the concept of flow diagnostics based on standard finite-volume discretizations.

Because of their low computational cost, flow diagnostics methods can easily be incorporated into interactive graphical tools that offer rapid and interactive screening and preprocessing capabilities. This, however, is not easy to present in book form, and if you want to see these tools in practice, you should try out some of the examples that follow MRST study the exercises presented throughout this chapter. Flow diagnostics techniques can also be utilized to post-process more comprehensive simulation methods and to perform what-if and sensitivity analyzes in parameter regions surrounding preexisting simulations. As such, flow diagnostics offers a computationally inexpensive complement and/or alternative to the use of full-featured multiphase simulations to provide flow information in various reservoir management workflows. .

13.1 Flow patterns and volumetric connections

You have already been introduced to the basic quantities that lie at the core of flow diagnostics in Chapters 4 and 5: As you probably recall from Section 4.3.3, we can derive time lines that show how heterogeneity affects flow patterns for an instantaneous velocity field \vec{v} by computing

- the forward time-of-flight, defined by

$$\vec{v} \cdot \nabla \tau_f = \phi, \quad \tau_f|_{\text{inflow}} = 0, \quad (13.1)$$

which measures time it takes a neutral particle to travel to a given point in the reservoir from the nearest fluid source or inflow boundary; and

- the backward time-of-flight, defined by

$$-\vec{v} \cdot \nabla \tau_b = \phi, \quad \tau_b|_{\text{outflow}} = 0, \quad (13.2)$$

which measures the time it takes a neutral particle to travel from any point in the reservoir to the nearest fluid sink or outflow boundary.

The sum of the forward and backward time-of-flight at a given point in the reservoir gives the total residence time of an imaginary particle as it travels from the nearest fluid source or inflow boundary to the nearest fluid sink or outflow boundary. Studying iso-contours of time-of-flight will give an indication of how more complex multiphase displacements may evolve under fixed well and boundary conditions, and will reveal more information about the flow field than pressure and velocities alone. This was illustrated already in Chapter 5, where Figure 5.3 on page 164 showed time lines for a quarter five-spot flow pattern and the total residence time was used to distinguish high-flow regions from stagnant regions. Likewise, in Figure 5.11 on page 177 we used time-of-flight to identify non-targeted regions for a complex field model, that is, regions with high τ_f values that were likely to remain unswept and hence were obvious targets to investigate for placement of additional wells. Time-of-flight can also be used to derive various measures of dynamic heterogeneity, as we will see in the next section, or to compute proxies of economical measures such as net-present value for models that contain multiphase fluid information, see [184].

In similar manner, we can determine the points in the reservoir that are affected by a given fluid source or inflow boundary by solving the following injector (or inflow) volumetric partition equation,

$$\vec{v} \cdot \nabla c_i^k = 0, \quad c_i^k|_{\text{inflow}} = 1. \quad (13.3)$$

To understand what this equation does, let us think of an imaginary painting experiment in which we inject a mass-less, non-diffusive ink of a unique color at each fluid source or point on the inflow boundary we want to trace the influence from. The ink will start flowing through the reservoir and paint every point it gets in contact with. Eventually, the fraction of different inks that flow past each point in the reservoir will reach a steady state, and by measuring these fractions, we can determine the extent to which each different ink influences a specific point. Likewise, to determine how much each point in the reservoir is influenced by a fluid sink or point on the outflow boundary, we can reverse the flow field and solve similar equations for producer (or outflow) partitions,

$$-\vec{v} \cdot \nabla c_p^k = 0, \quad c_p^k|_{\text{outflow}} = 1. \quad (13.4)$$

To summarize, the basic computation underlying flow diagnostics consists of three parts: (i) solution of a pressure equation to determine the bulk fluid movement; (ii) solution of a set of numerical equations to partition the model into volumetric flow regions; and (iii) solution of time-of-flight equations to give time lines that describe the flow within each region. In the following, we will describe in more detail how these basic quantities can be combined and processed to provide more insight into flow patterns and volumetric connections in the reservoir.

13.1.1 Volumetric partitions

If all parts of the inflow (or outflow) are assigned a unique 'color' value, the resulting color concentration should in principle produce a partition of unity for all parts of the reservoir that are in communication with the inflow (or outflow) boundary. In practice, one may not be able to obtain an exact partition of unity because of numerical errors. Based on the inflow and outflow partitions, we can further define

- drainage regions – each such region represents the reservoir volume that eventually will be drained by a given producer (or outflow boundary) given that the current flow field \vec{v} prevails until infinity;
- sweep regions – each such region represents the reservoir volume that eventually will be swept by a given injector (or inflow boundary) if the current flow conditions remain forever;
- well pairs – pairs of injectors and producers that are in communication with each other;
- well-pair regions – regions of the reservoir in which the flow between a given injector and producer takes place.

Drainage and sweep regions are typically determined by a majority vote over all 'colors', while well pairs are determined by finding all injectors whose concentration is positive in one of the well completions of a given producer (or vice versa). Well-pair regions can be found by intersecting drainage and sweep regions, or alternatively by intersecting injector and producer partitions. Well-allocation factors will be discussed in more detail in Section 13.1.3 and in Chapter 15 in conjunction with single-phase upscaling.

Our default choice would be to assign a unique 'color' to each injector and producer, but you can also subdivide some of the wells into multiple segments and trace the influence of each segment separately. This can, for instance, be used to determine if a (horizontal) well has cross-flow, so that fluid injected in one part of the well is drawn back into the wellbore in another part of the well, or fluids produced in one completion is pushed out again in another.

We have already encountered the function `computeTimeOfFlight` for computing time-of-flight in Section 5.3. The `diagnostics` module offers an additional utility function

```
D = computeTOFandTracer(state, G, rock, 'wells', W, ...)
```

that computes forward and backward time-of-flight, injector and producer coloring, as well as sweep and drainage regions in one go for models with flow driven by wells. These quantities are represented as fields in the structure D:

- `inj` and `prod` give the indices for the injection and production wells in the well structure `W`;
- `tof` is a $2 \times n$ vector with τ_f in its first and τ_b in its second column;
- `itracer` and `ptracer` contains the color concentrations for the injectors and producers;

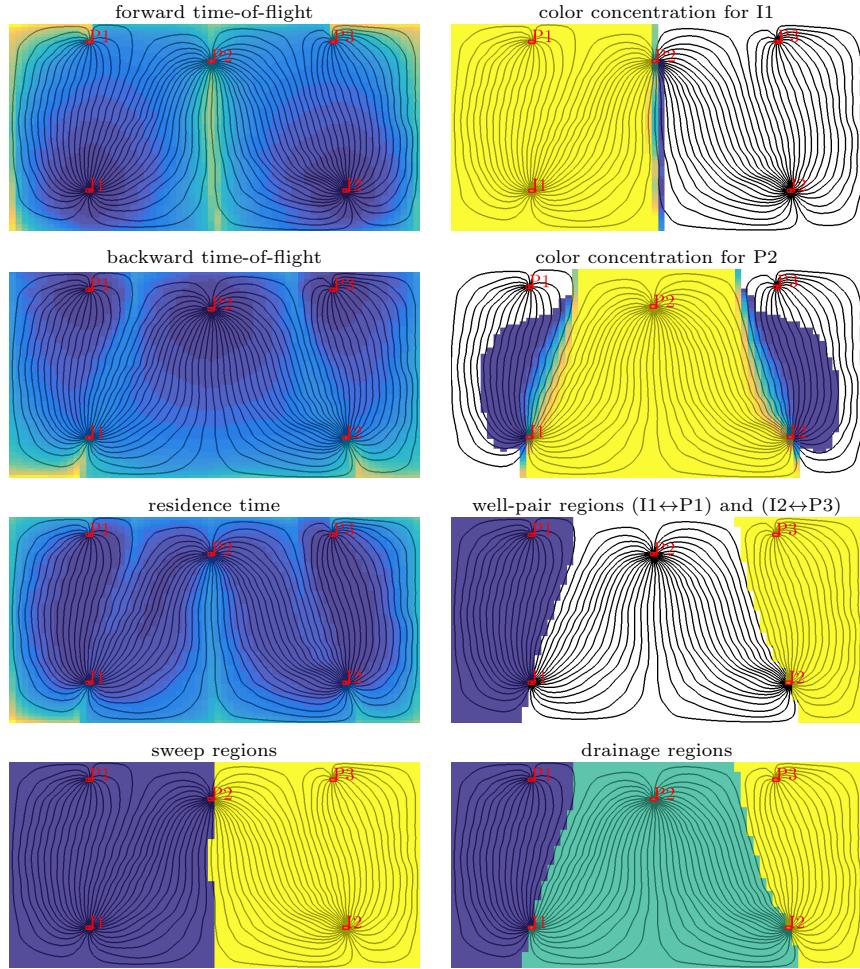


Fig. 13.1. Time-of-flight, 'color' concentration, and various types of volumetric delineations for a simple case with two injectors and three producers. The source code necessary to generate the plots is given in the `showDiagnostBasic` tutorial in the book module.

- `iport` and `pport` hold the partitions resulting from a majority vote.

Similar quantities can be associated with boundary conditions and/or source terms, but support for this has not yet been implemented in MRST. Well pairs can be identified by

```
WP = computeWellPairs(G, rock, W, D)
```

which also computes the pore volume of the region of the reservoir associated with each pair.

Figure 13.1 shows time-of-flight, color concentrations, volumetric partitions and a few combinations thereof for a simple flow problem with two injectors and three production wells. There are many other ways these basic quantities can be combined and plotted to reveal volumetric connection and provide enhanced insight into flow patterns. One can, for instance, combine sweep regions with time-of-flight to provide a simple forecasts of contacted volumes or to visualize how displacement fronts move through the reservoir. To this end, one would typically also include estimates of characteristic wave-speeds from multiphase displacement theory.

13.1.2 Time-of-flight per partition region: improved accuracy

It is important to understand that the time-of-flight values computed by a finite-volume method like the one discussed in Section 4.4.3 are *volume-average values*, which cannot be compared directly with the *point values* one obtains by tracing streamlines as described in Section 4.3.3. Since time-of-flight is a quantity defined from a global line integral (see (4.38)), the point-wise variations can be large inside a single grid cell, in particular near flow divides and in the near-well regions, where both high-flow and low-flow streamlines converge. To improve the accuracy of the time-of-flight within each well-pair region, one can use the color concentrations to recompute the time-of-flight values for each partition region,

$$\vec{v} \cdot \nabla(c_i^k \tau_f^k) = c_i^k \phi. \quad (13.5)$$

In MRST, this is done by passing the option `computeWellTOFs` to the time-of-flight solver, using a call that looks something like

```
T = computeTimeOfFlight(state, G, rock, 'wells', W, ...
    'tracer', {W(inj).cells}, 'computeWellTOFs', true);
```

which will then append one extra column for each color at the end of the return parameter T.

13.1.3 Well-allocation factors

Apart from a volumetric partition of the reservoir, one is often interested in knowing how much of the inflow to a given producer can be attributed to each of the injectors, or conversely, how the 'push' from a given injector is distributed to the different producers. We will refer to this as well-allocation factors, which can be further refined so that they also describe the cumulative flow from the toe to the heel of a well. By computing the cumulative flux from the toe to the heel of the well and plotting this flux as a function of the distance from the toe (with the flux on the x -axis and distance on the y -axis) we get a plot that is reminiscent of the plot from a production-logging tool.

To formally define the well-allocation factors, we use the notation from Section 4.4.2 on page 140, so that $\mathbf{x}[c]$ denotes the value of vector \mathbf{x} in cell c . Next, we let \mathbf{c}_n^i denote the injector color concentration associated with well (or well segment) number n , let \mathbf{c}_m^p denote the producer concentration associated with well number m , \mathbf{q} the vector of well fluxes, and $\{w_k^n\}_k$ the cells in which well number n is completed. Then, the cumulative factors are defined as

$$\begin{aligned} a_{nm}^i[w_\ell^n] &= \sum_{k=1}^\ell \mathbf{q}[w_k^n] \mathbf{c}_m^p[w_k^n], \\ a_{mn}^p[w_\ell^m] &= \sum_{k=1}^\ell \mathbf{q}[w_k^m] \mathbf{c}_n^i[w_k^m]. \end{aligned} \quad (13.6)$$

The total well-allocation factor equals the cumulative factor evaluated at the heel of the well. Well-allocation factors are computed using the function `computeWellPair` and are found as two arrays of structs, `WP.inj` and `WP.prod`, that give the allocation factors for all the injection and production wells (or segments) accounted for in the flow diagnostics. In each struct, the array `alloc` gives the a_{nm} factors, whereas the influx or outflux that cannot be attributed to another well or segment is represented in the array `ralloc`.

13.2 Measures of dynamic heterogeneity

Whereas primary recovery can be reasonably approximated using averaged petrophysical properties, secondary and tertiary recovery is strongly governed by the intrinsic variability in rock properties and geological characteristics. This variability, which essentially can be observed at all scales in the porous medium, is commonly referred to as ‘heterogeneity’. As we have seen in previous chapters, both the rock’s ability to store and to transmit fluids are heterogeneous. However, it is the heterogeneity in permeability that has the most pronounced effect on flow patterns and volumetric connections in the reservoir. The importance of heterogeneity has been recognized from the earliest days of petroleum production, and over the years a number of static measures have been proposed to characterize heterogeneity, such as flow and storage capacity, Lorenz coefficient, Koval factor, and Dykstra–Parson’s permeability variation coefficient, to name a few; see e.g., [145] for a more comprehensive overview.

In this section, we will show how some of the static heterogeneity measures from classical sweep theory can be reinterpreted in a dynamic setting if we calculate them from the time-of-flight (and color concentrations) associated with an instantaneous flow field [232]. Static measures describe the spatial distribution of permeability and porosity, and large static heterogeneity means that there are large (local) variations in the rock’s ability to store and transmit fluids. Dynamic heterogeneity measures, on the other hand, describe the distribution of flow-path lengths and connection structure, and large heterogeneity values show that there are large variations in travel and residence

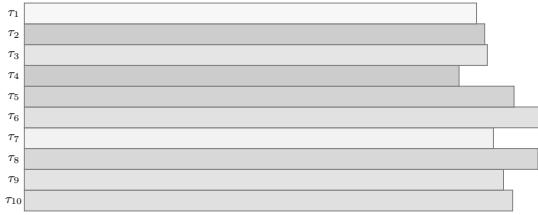


Fig. 13.2. Streamtube analogue used to define dynamic flow and storage capacity. Colors illustrate different average porosities.

times, which again tend to manifest itself in early breakthrough of injected fluids. Experience has shown that these measures, and particularly the dynamic Lorenz coefficient, correlates very well with forecasts of hydrocarbon recovery predicted by more comprehensive flow simulations and hence can be used as effective flow proxies in various reservoir management workflows, see [232, 230, 184].

13.2.1 Flow and storage capacity

A computation of forward and backward time-of-flight is the starting point for defining dynamic heterogeneity measures. We start by defining dynamic flow and storage capacity. To this end, we can think of the reservoir as a set of N streamtubes (non-communicating volumetric flow paths) that each has a volume V_i , a flow rate q_i , and a residence time $\tau_i = V_i/q_i$. The streamtubes are sorted so that their residence times are ascending, $\tau_1 \leq \tau_2 \leq \dots \leq \tau_N$, see Figure 13.2. Inside each streamtube, we assume a piston type displacement; think of a blue fluid pushing a red fluid from the left to the right in the figure. We then define the normalized flow capacity F_i and storage capacity Φ_i by,

$$\Phi_i = \sum_{j=1}^i V_j / \sum_{j=1}^N V_j, \quad F_i = \sum_{j=1}^i q_j / \sum_{j=1}^N q_j. \quad (13.7)$$

Here, Φ_i is the volume fraction of all streamtubes that have 'broken through' at time τ_i and F_i represent the corresponding fractional flow, i.e., the fraction of the injected fluid to the total fluid being produced. These two quantities can be plotted in a diagram as shown in Figure 13.3. From this diagram, we can also define the fractional recovery curve defined as the ratio of inplace fluid produced to the total fluid being produced; that is $(1 - F)$ plotted versus dimensionless time $t_D = d\Phi/dF$ measured in units of pore volumes injected.

To see how the F - Φ diagram can be seen as a measure of dynamic heterogeneity, we first consider the case of a completely homogeneous displacement, for which all streamtubes will break through at the same time τ . This means that, $(\Phi_i - \Phi_{i-1})/(F_i - F_{i-1}) \propto V_i/q_i$ is constant, which implies that $F = \Phi$, since both F and Φ are normalized quantities. Next, we consider a heterogeneous displacement in which all the streamtubes have the same flow rate q .

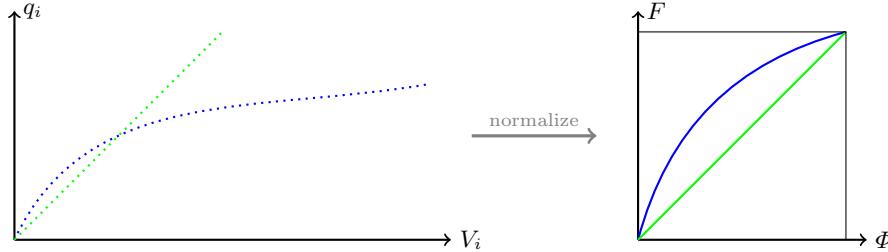


Fig. 13.3. Construction of the $F\text{-}\Phi$ diagram. The plot to the left shows flow rates q_i plotted as function of streamtube volumes V_i for a homogeneous displacement (green) and for a heterogeneous displacement (blue). The right plot shows the corresponding $F\text{-}\Phi$ diagrams, where the flow rates and the streamtube volumes have been normalized.

Since the residence times τ_i form a monotonically increasing sequence, we have that $\{V_i\}$ will also be monotonically increasing. In general, $F(\Phi)$ is a concave function, where the steep initial slope corresponds to high-flow regions giving early breakthrough, whereas the flat trailing tail corresponds to low-flow and stagnant regions.

In the continuous case, we can (with a slight abuse of notation) define the storage capacity as

$$\Phi(\tau) = \int_0^\tau \phi(\vec{x}(s)) ds \quad (13.8)$$

where $\vec{x}(\tau)$ represents all streamlines whose total travel time equals τ . By assuming incompressible flow, we have that pore volume equals the flow rate times the residence time, $\phi = q\tau$, and hence we can define the flow capacity as

$$F(\tau) = \int_0^\tau q(\vec{x}(s)) ds = \int_0^\tau \frac{\phi(\vec{x}(s))}{s} ds. \quad (13.9)$$

From this, we can define *normalized, dynamic* flow and storage capacities by

$$\hat{\Phi}(\tau) = \frac{\Phi(\tau)}{\Phi(\infty)}, \quad \hat{F}(\tau) = \frac{F(\tau)}{F(\infty)}.$$

Henceforth, we will only discuss the normalized quantities and for simplicity we will also drop the hat symbol. To compute these quantities in practice, one would then have to first compute a representative set of streamlines, associate a flow rate, a pore volume, and a total travel time to each streamline, and then compute the cumulative sums as in (13.7).

Next, we consider how these concepts carry over to our grid setting where time-of-flight is computed by a finite-volume method and not by tracing streamlines. Let pv be an $n \times 1$ array containing the pore volume of the n cells in the grid and tof be an $n \times 2$ array containing the forward and backward time-of-flights. We can now compute the cumulative, normalized storage capacity Phi as follows:

```
t      = sum(tof,2); % total travel time
[ts,ind] = sort(t); % sort cells based on travel time
v      = pv(ind); % put pore volumes in correct order
Phi    = cumsum(v); % cumulative sum
vt     = full(Phi(end)); % total volume of region
Phi    = [0; Phi/vt]; % normalize to units of pore volumes
```

In our finite-volume formulation, we do not have direct access to the flow rate for each cell, but this can easily be computed as the ratio between pore volume and residence time if we assume incompressible flow. With this, it is straightforward to compute the cumulative, normalized flow capacity F

```
q      = v./ts; % back out flux based on incompressible flow
ff    = cumsum(q); % cumulative sum
ft    = full(ff(end)); % total flux computed
F     = [0; ff/ft]; % normalize and store flux
```

This is essentially what is implemented in the utility function `computeFandPhi` in the `diagnostics` module.

The result of the above calculation is that we have two sequences Φ_i and F_i that are both given in terms of the residence time τ_i . If we sort the points (Φ_i, F_i) according to ascending values of τ_i , we obtain a sequence of discrete points that describe a parametrized curve in 2D space. The first end-point of this curve is at the origin: If no fluids have entered the domain, the cumulative flow capacity is obviously zero. Likewise, full flow capacity is reached when the domain is completely filled, and since we normalize both Φ and F by their value at the maximum value of τ , this corresponds to the point (1,1). Given that both F and Φ increase with increasing values of τ , we can use linear interpolation to define a continuous, monotonic, increasing function $F(\Phi)$.

13.2.2 Lorenz coefficient and sweep efficiency

The Lorenz coefficient is a popular measure of heterogeneity, and is defined as the difference in flow capacity from that of an ideal piston-like displacement:

$$L_c = 2 \int_0^1 (F(\Phi) - \Phi) d\Phi, \quad (13.10)$$

In other words, the Lorenz coefficient is equal twice the area under the $F(\Phi)$ curve and above the line $F = \Phi$, and has values between zero for homogeneous displacement and unity for an infinitely heterogeneous displacement, see Figure 13.4. Assuming that the flow and storage capacity are given as two vectors `F` and `Phi`, the Lorenz coefficient can be computed by applying a simple trapezoid rule

```
v  = diff(Phi,1);
Lc = 2*(sum((F(1:end-1)+F(2:end))/2.*v) - .5);
```

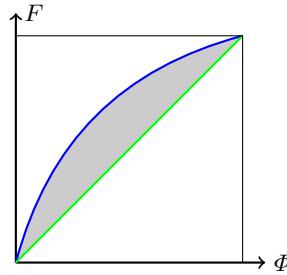


Fig. 13.4. The definition of the Lorenz coefficient from the F - Φ diagram. The green line represents a completely homogeneous displacement in which all flow paths have equal residence times, where the blue line is a heterogeneous displacement in which there is variation in the residence times. The Lorenz coefficient is defined as two times the gray area.

which is implemented in the `computeLorenz` function in the `diagnostics` module.

The F - Φ diagram can also be used to compute the *volumetric sweep efficiency* E_v , which measures how efficient injected fluids are used. Here, E_v is defined as the volume fraction of inplace fluid that has been displaced by injected fluid, or equivalently, as the ratio between the volume contacted by the displacing fluid at time t and the volume contacted at time $t = \infty$. In our streamtube analogue, only streamtubes that have not yet broken through will contribute to sweep the reservoir. Normalizing by total volume, we thus have

$$\begin{aligned} E_v(t) &= \frac{q}{V} \int_0^t [1 - F(\Phi(\tau))] d\tau \\ &= \frac{qt}{V} - \frac{q}{V} \int_0^t F(\tau) d\tau = \frac{qt}{V} - \frac{q}{V} \left[F(t)t - \int_0^F \tau dF \right] \\ &= \frac{t}{\bar{\tau}} (1 - F(t)) + \frac{1}{\bar{\tau}} \int_0^{\Phi} \bar{\tau} d\Phi = \Phi + (1 - F) \frac{d\Phi}{dF} = \Phi + (1 - F)t_D \end{aligned}$$

The third equality follows from integration by parts, and the fourth equality since $\bar{\tau} = V/q$ and $\tau dF = \bar{\tau} d\Phi$. Here, the quantity $d\Phi/dF$ takes the role as dimensionless time. Prior to breakthrough, $E_v = t_D$. After breakthrough, Φ is the volume of fully swept flow paths, whereas $(1 - F)t_D$ is the volume of flow paths being swept.

The implementation in MRST is quite simple and can be found in the utility function `computeSweep`. Starting from the two arrays `F` and `Phi`, we first remove any flat segments in `F` to avoid division by zero

```

inz      = true(size(F));
inz(2:end) = F(1:end-1) ~= F(2:end);
F      = F(inz);
Phi    = Phi(inz);

```

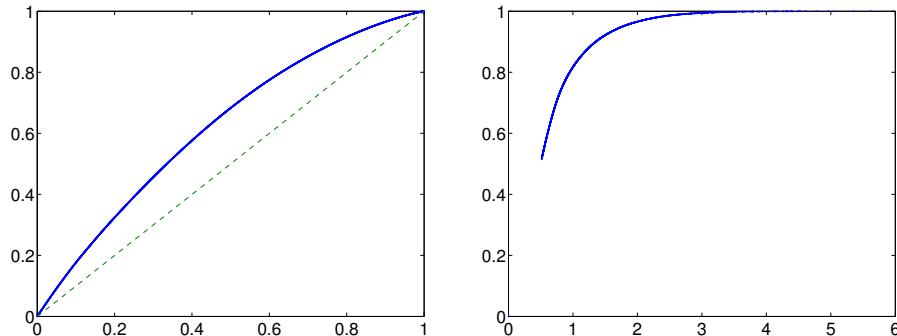


Fig. 13.5. $F-\Phi$ and sweep diagram for the simple case with two injectors and three producers, which has a Lorenz coefficient of 0.2475.

Then, dimensionless time and sweep efficiency can be computed as follows

```
tD = [0; diff(Phi)./diff(F)];
Ev = Phi + (1-F).*tD;
```

Figure 13.5 shows the $F-\Phi$ and the sweep diagram for the simple example with two injectors and three producers from Figure 13.1. For this particular setup, the Lorenz coefficient was approximately 0.25, which indicates that we can expect a mildly heterogeneous displacement with some flow paths that break through early and relatively small stagnant regions. From the diagram of the sweep efficiency, we see that 70% of the fluids in-place can be produced by injecting one pore volume and by injecting two additional pore volumes almost all the in-place fluid can be produced.

13.2.3 Summary of diagnostic curves and measures

Altogether, we have defined three different curves that can be derived from the residence/travel time. The curves shown in Figure 13.6 are visually intuitive and emphasize different characteristics of the displacement:

- The $F-\Phi$ curve is useful for assessing the overall level of displacement heterogeneity. The closer this curve is to a straight line, the better is the displacement.
- The fractional recovery curve emphasizes early-time breakthrough behavior and can have utility as a proxy for fractional recovery of the fluid in-place.
- The sweep efficiency highlights the behavior after breakthrough and has utility as a proxy for recovery factor.

The curves can be defined for the field as a total, or be associated with sector models, individual swept volumes, well-pair regions, and so on.

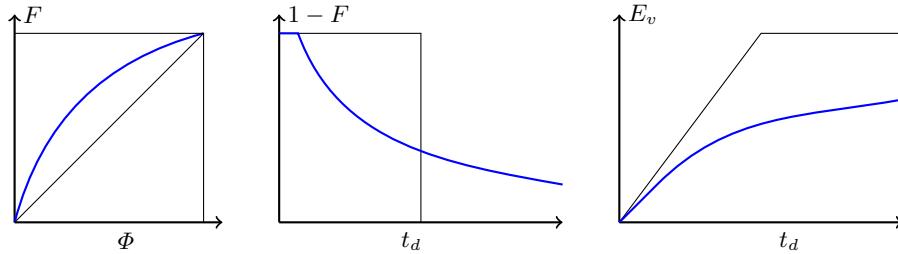


Fig. 13.6. The three basic flow diagnostics curves: F - Φ diagram, fractional recovery curve, and sweep efficiency. All quantities Φ , F , E_v , and t_D are dimensionless; t_D is given in terms of pore volumes injected (PVI).

Whereas visually intuitive information is useful in many workflows, others need measures defined in terms of real numbers. In our work, we have primarily used the Lorenz coefficient. However, dynamic analogues of other classical heterogeneity measures can be defined in a similar fashion. For instance, the dynamic Dykstra–Parsons’ coefficient is defined as

$$V_{DP} = \frac{(F')_{\Phi=0.5} - (F')_{\Phi=0.841}}{(F')_{\Phi=0.5}}, \quad (13.11)$$

where $\Phi = 0.5$ corresponds to the mean storage capacity, while $\Phi = 0.841$ is the mean value plus one standard deviation. Likewise, we can define the dynamic flow heterogeneity index,

$$F_{HI} = F(\Phi^*)/\Phi^*, \quad F'(\Phi^*) = 1, \quad (13.12)$$

Here, one can show that $(\frac{dF}{d\Phi})_i = \frac{\bar{\tau}}{\tau_i}$, where $\bar{\tau} = V/q = \sum V_i / \sum q_i$ is the average residence time for all the streamtubes. These heterogeneity measures have not yet been implemented in the `diagnostics` module.

COMPUTER EXERCISES:

86. Implement Dykstra–Parsons’ coefficient (13.11) and the flow heterogeneity index (13.12).
87. Use time-of-flight values defined per color region as discussed in Section 13.1.2 on page 502 to implement refined versions of the dynamic heterogeneity measures.
88. Compute heterogeneity measures for each well pair in the model with two injectors and three producers. (Original source code: `showDiagnostBasics`) Are there differences between the different regions?
89. Try to make the displacement shown in Figures 13.1 and 13.5 less heterogeneous by moving the wells and/or by changing the relative magnitude of the injection/production rates.

13.3 Case studies

The use of flow diagnostics is best explained through examples. In this section we therefore go through several cases and demonstrate various ways in which flow diagnostics can be used to enhance our understanding of flow patterns and volumetric connections, tell us how to change operational parameters such as well placement and well rates to improve recovery, etc.

13.3.1 Tarbert formation: volumetric connections

As our first example, we consider a subset of the SPE 10 data set consisting of the top twenty layers of the Tarbert formation, see Section 2.5.3. We modify the original inverted five-spot well pattern by replacing the central injector by two injectors that are moved a short distance from the model center (see Figure 13.7), assume single-phase incompressible flow, and solve the corresponding flow problem. A complete description of the setup can be found in `showWellPairsSPE10.m` in the book module.

Given the geological model represented in terms of the structures `G` and `rock`, the wells represented by `W`, and the reservoir state by `rS`, we first compute the time-of-flight and volumetric partitions:

```
D = computeTOFandTracer(rS, G, rock, 'wells', W);
```

This gives us the information we need to partition the volume into different drainage and sweep volumes. The simplest way to do this for the purpose of visualization is to use a majority vote over the injector and producer partitions to determine the well that influences each cell the most as shown in Figure 13.8. The result of this majority vote is collected in `D.ppart` and `D.ipart`, respectively, and the essential commands to produce the two upper plots in Figure 13.8 are:

```
plotCellData(G,D.ipart, ...);
plotCellData(G,D.ppart,D.ppart>1, ...);
```

Since there are two injectors, we would expect to see two different sweep regions. However, in the figure there is also a small blue volume inside the triangular section bounded by I1, P1, and P2, which corresponds to an almost impermeable part of the reservoir that will not be swept by any of the injectors. The well pattern is symmetric and for a homogeneous medium we would therefore expect that the two pressure-controlled injectors would sweep symmetric volumes of equal size. For the highly heterogeneous Tarbert formation, however, the sweep regions are quite irregular and clearly not symmetric. Because of the two wells are completed in cells with very different permeability, the injection rate of I2 is approximately six times that of I1, and hence I2 will sweep a much larger region than I1. In particular, we see that I2 is the injector that contributes most to flooding the lower parts of the region near P3,

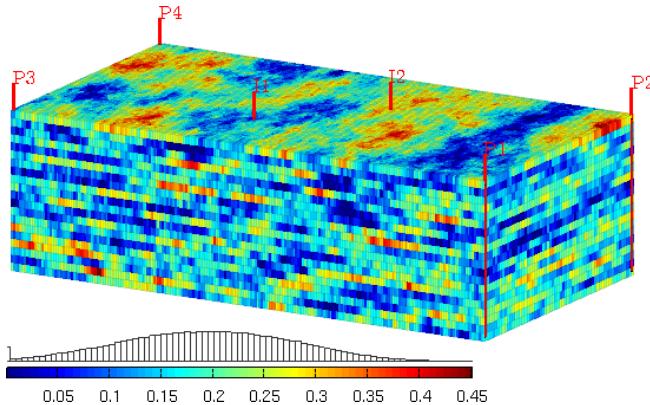


Fig. 13.7. Porosity and well positions for a model consisting of subset of the Tarbert formation in Model 2 from the 10th SPE Comparative Solution Project

even though I1 is located closer. Looking at the drainage and sweep regions in conjunction, it does not seem likely that I1 will contribute significantly to support the production from wells P1 and P2 unless we increase its rate.

When using a majority vote to determine drainage and sweep regions, we disregard the fact that there are regions that are influenced by more than one well. To visualize such regions of the reservoir, we can blend in a gray color in all cells in which more than one color has nonzero concentration as shown in the bottom plot of Figure 13.8. The plot is generated by the following call:

```
plotTracerBlend(G, D.ppart, max(D.ptracer, [], 2), ...);
```

Having established the injection and producer partitions, we can identify well pairs and compute the pore volumes of the region associated with each pair:

```
WP = computeWellPairs(rS, G, rock, W, D);
pie(WP.vols, ones(size(WP.vols)));
legend(WP.pairs, 'location', 'Best');
```

To visualize the volumetric regions, we compute the tensor product of the injector and producer partitions and then compress the result to get a contiguous partition vector with a zero value signifying unswept regions:

```
p = compressPartition(D.ipart + D.ppart*max(D.ipart))-1;
plotCellData(G,p,p>0, 'EdgeColor', 'k', 'EdgeAlpha', .05);
```

The result is shown in Figure 13.9, and confirms our previous observations of the relative importance of I1 and I2. Altogether, I1 contributes to sweep approximately 16% of the total pore volume, shown as the light red and the yellow regions in the 3D plot.

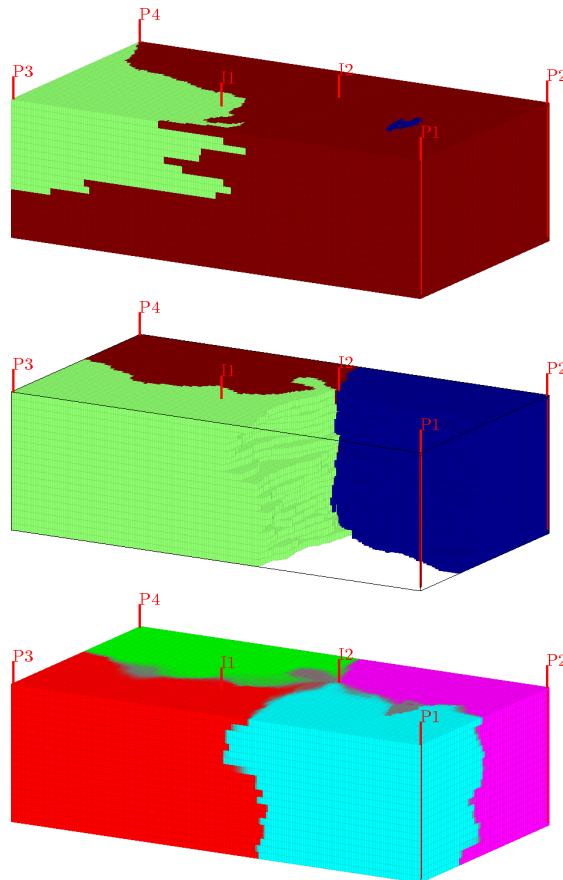


Fig. 13.8. Sweep (top) and drainage regions (middle) determined by a majority vote over injector and producer partitions, respectively, for the Tarbert model. The bottom plot shows a refined partition in which gray color signifies regions that are affected by multiple colors.

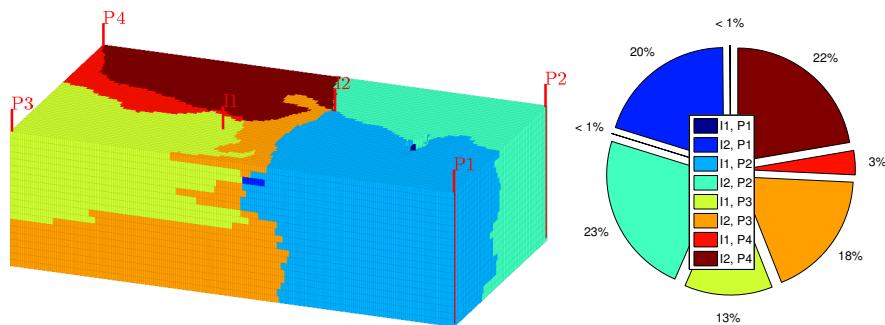


Fig. 13.9. Well-pair regions and associated fraction of the total pore volume for the upper twenty layers of the Tarbert formation.

It is also interesting to see how these volumetric connections affect the fluxes in and out of wells. To this end, we should look at the cumulative well-allocation factors, which are defined as the cumulative flux in/out of a well from bottom to top perforation of a vertical well, and from toe to heel for a deviated well. We start by computing the flux allocation manually for the two injectors (outflow fractions are given from well head and downward and hence need to be flipped):

```
for i=1:numel(D.inj)
    subplot(1,numel(D.inj),i); title(W(D.inj(i)).name);
    alloc = cumsum(flipud(WP.inj(i)).alloc,1);
    barh(flipud(WP.inj(i).z), alloc, 'stacked'); axis tight
    lh = legend(W(D.prod).name,4);
    set(gca, 'YDir', 'reverse');
end
```

Figure 13.10 shows the resulting bar plots of the cumulative allocation factors. These plots confirm and extend the understanding we have developed by studying volumetric connections: I1 will primarily push fluids towards P3. Some fluids are also pushed towards P4, and we also observe that there is almost no outflow in the top three perforations where the rock has low quality. Injection from I2, on the other hand, contributes to uphold the flux into all four producers. We also see that the overall flux is not well balanced. Producer P1 has significantly lower inflow than the P2 to P4. Alternatively, we can use the library functions `plotWellAllocationPanel(D, WP)` from the `diagnostics` module to compute and visualize the well-allocation factors for all the wells in the model, as shown in Figure 13.11.

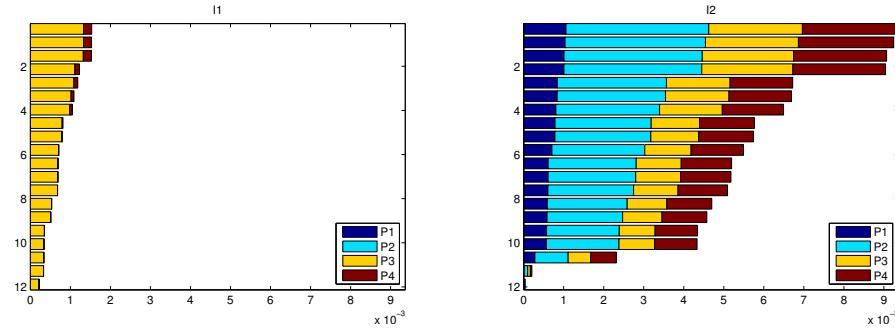
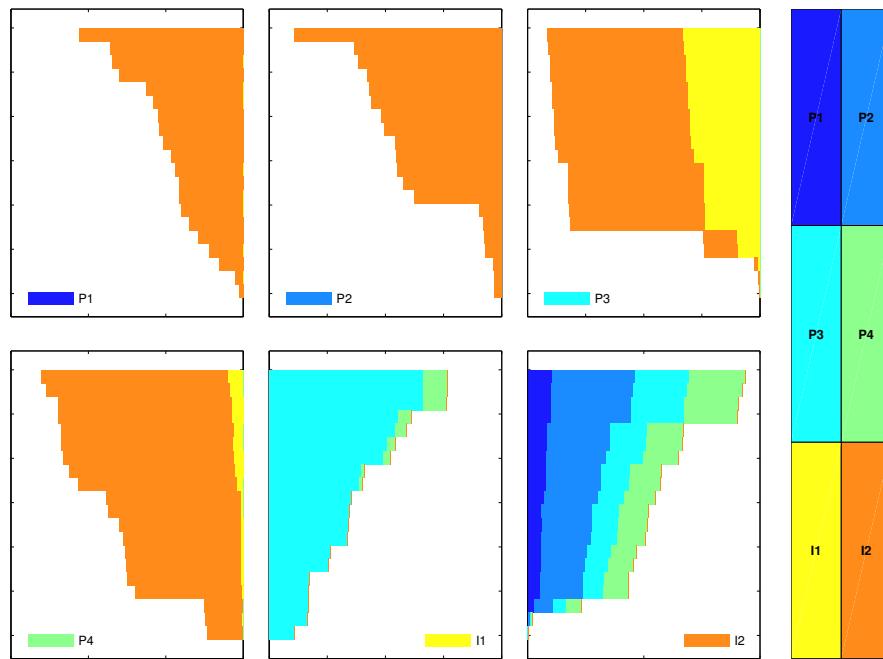
Finally, to look more closely at the performance of the different completions along the well path, we can divide the completion intervals into bins and assign a corresponding set of pseudo-wells for which we recompute flow diagnostics. As an example, we split the completions of I1 into three bins and the completions of I2 into four bins.

```
[rSp,Wp] = expandWellCompletions(rS,W,[5, 3; 6, 4]);
Dp = computeTOFandTracer(rSp, G, rock, 'wells', Wp);
```

Figure 13.12 shows the majority-voted sweep regions for the four segments of I2; to better see the various sweep regions, the 3D plot is rotated 180 degrees compared with the other 3D plots of this model. To obtain the figure, we used the following key statements

```
plotCellData(G, Dp.ipart, (Dp.ipart>3) & (Dp.ipart<8), ...);
WPp = computeWellPairs(rSp, G, rock, Wp, Dp);
avols = accumarray(WPp.pairIx(:,1),WPp.vol);
pie(avols(4:end));
```

Notice, in particular, that fluids injected in the lowest segment is the major contributor in almost half of the well's total sweep region.

**Fig. 13.10.** Well-allocation factors for the two injectors of the Tarbert model.**Fig. 13.11.** Normalized well-allocation factors for all wells of the Tarbert model.

13.3.2 Layers of SPE 10: heterogeneity and sweep improvement

In this example, we first compute the Lorenz coefficient for all layers of the SPE 10 model subject to an inverted five-spot well pattern. We then pick one of the layers and show how we can balance the well allocation and improve the Lorenz coefficient and the areal sweep by moving some of the wells to regions with better sand quality.

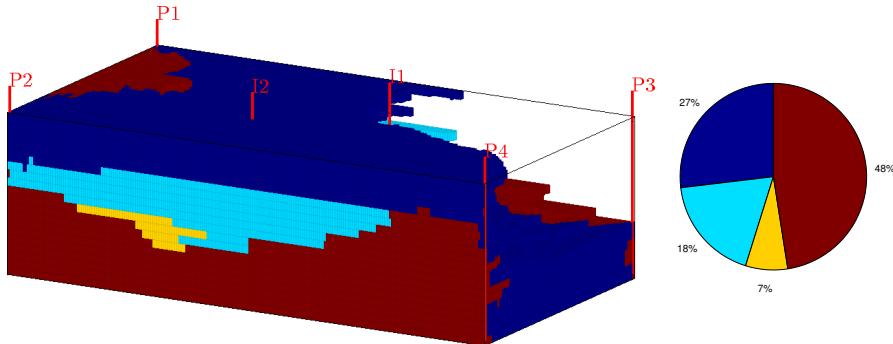


Fig. 13.12. Majority-voted sweep regions for I2 of the Tarbert case divided into four segments that each are completed in five layers of the model. (Notice that the view angle is rotate 180 degrees compared with Figure 13.8.) The pie chart shows the fraction of the total sweep region that can attributed to each segment of the well.

To compute Lorenz coefficient for all layers in the SPE 10 model, we first define a suitable $60 \times 220 \times 1$ grid covering a rectangular area of $1200 \times 2200 \times 2$ ft³. Then, we loop over all the 85 layers using the following essential lines:

```

for n=1:85
    rock = getSPE10rock(1:cartDims(1),1:cartDims(2),n);
    rock.perm = convertFrom(rock.perm, milli*darcy);
    rock.poro = max(rock.poro, 1e-4);

    W = [];
    for w = 1:numel(wtype),
        W = verticalWell(..);
    end

    T = computeTrans(G, rock);
    rS = incompTPFA(initState(G, W, 0), G, T, fluid, 'wells', W);

    D      = computeTOFandTracer(rS, G, rock, 'wells', W, 'maxTOF', inf);
    [F,Phi] = computeFandPhi(poreVolume(G,rock), D.tof);
    Lc(n)   = computeLorenz(F,Phi);
end

```

Because the permeability changes for each layer, we need to recompute the transmissibility. Likewise, we regenerate the well objects to ensure correct well indices when updating the petrophysical data. Complete source code can be found in the script `computeLorenzSPE10` in the book module.

Figure 13.13 reports the Lorenz coefficients for all layers for a setup in which both the injector and the four producers are controlled by bottom-hole pressure. We relatively large dynamic heterogeneity and the variation among

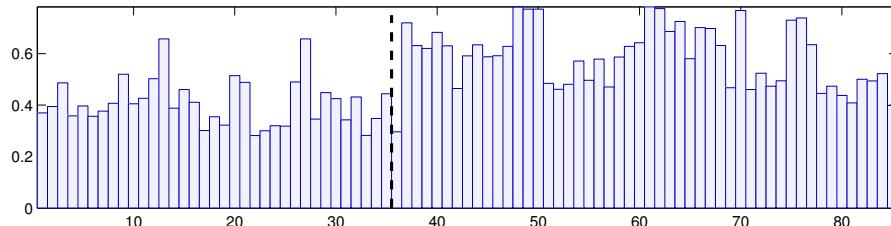


Fig. 13.13. Lorenz coefficient for each of the 85 horizontal layers of the SPE 10 model subject to an inverted five-spot pattern with injector and producers controlled by bottom-hole pressure. The dashed line shows the border between the Tarbert and the Upper Ness layers.

individual layers within the same formation is a result of our choice of well controls. The actual injection and production rates achieved with pressure-controlled wells are very sensitive to each well being perforated in a region of good sand quality. Good sand quality is difficult to ensure when using fixed well positions, and hence pressure-controlled wells will generally accentuate heterogeneity effects. To see this, you should modify the script and rerun the case with equal rates in all the four producers.

Our somewhat haphazard placement of the wells is not what a good reservoir engineer would recommend, but serves well to illustrate our next point. Since the Lorenz coefficient generally is quite large, most of the cases would have suffered from early breakthrough if we were to use this initial well placement for multiphase fluid displacement. Let us therefore pick one of the layers and see if we can try to improve the Lorenz coefficient and hence also the sweep efficiency. Figure 13.14 shows the well allocation and the sand quality near the production wells for Layer 61, which is the layer giving the worst Lorenz coefficient. The flux allocation shows that we have a very unbalanced displacement pattern where producer P4 draws 94.5% of the flux from the injector and producers P1 and P2 together draw only 1.2%. This can be explained if by looking at the sand quality in our reservoir. Producer P1 is completed in a low-quality sand and will therefore achieve a low rate if all producers operate at the same bottom-hole pressure. Producers P2 and P3 are perforated in cells with better sand, but are both completely encapsulated in regions of low sand quality. On the other hand, producer P4 is connected to the injector through a relatively contiguous region of high-permeable sand. Likewise, the largely concave $F\text{-}\Phi$ diagram shown to the left in Figure 13.15 testifies that the displacement is strongly heterogeneous and hence be characterized by large differences in the residence time of different flow paths, or in other words, suffer from early breakthrough of the displacing fluid. Hence, we would need to large amounts of the displacing fluid to recover the hydrocarbons from the low-quality sand; this can be seen from the weakly concave sweep diagram to the right in Figure 13.15. Altogether, we should expect a very unfavorable volumetric sweep from this well placement.

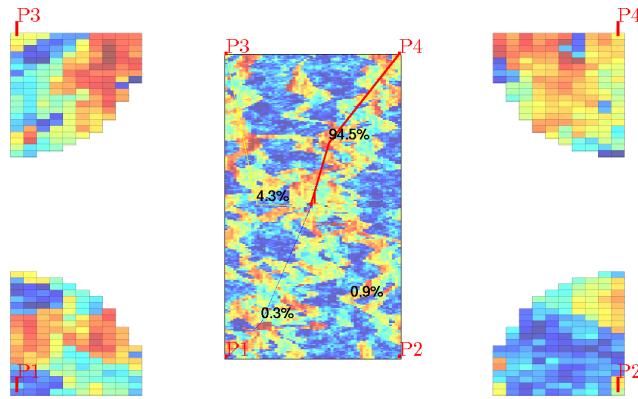


Fig. 13.14. Well configuration and flux allocation for the four well pairs with initial well configuration for Layer 61. (Red colors are good sands, while blue colors signify sands of low permeability and porosity.)

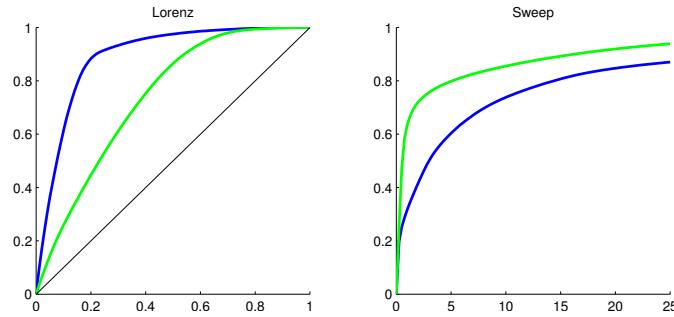


Fig. 13.15. $F\text{-}\Phi$ and sweep diagrams before (blue line) and after (green line) producers P1 to P3 have been moved to regions with better sand quality. Moving well reduces the Lorenz coefficient from 0.78 to 0.47.

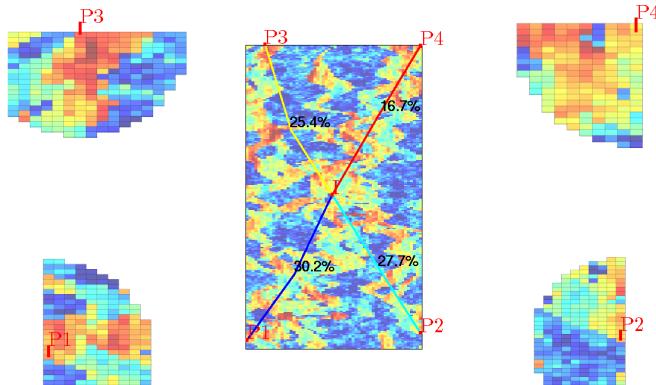


Fig. 13.16. Well configuration and flux allocation for the four well pairs after the producers P1 to P3 have been moved to regions with better sand quality.

Table 13.1. Volumetric flow rates in units $10^{-5} \text{ m}^3/\text{s}$ for each of the wells in Layer 61 of the SPE 10 model.

Placement	P1	P2	P3	P4	I
Initial	-0.0007	-0.0020	-0.0102	-0.2219	0.2348
Improved	-0.0719	-0.0660	-0.0604	-0.0398	0.2381

To improve the displacement, we can try to move each of the producers to a better location; that is, we should look for cells in the vicinity of each well that have better sand quality (higher porosity and permeability) *and* are connected to the injector by more contiguous paths of good quality sands. Figure 13.16 shows the well allocation after we have made such a move of producers P1, P2, and P3. Producer P4 was already in a good location, so we do not move it. Compared with our previous setup, the well allocation is now much more balanced and which is also confirmed by the well rates reported in Table 13.1. (Notice also that the overall reservoir rate has increased slightly). In Figure 13.15, we see that the $F\text{-}\Phi$ diagram has become significantly less concave and, likewise, that the sweep diagram has become much more concave. This testifies that the variation in residence times associated with different flow paths is much smaller and we should expect a more efficient and less heterogeneous volumetric sweep.

These types of dynamic heterogeneity measures are generally easy to use as a guide when searching for optimal or improved well placement. Because of their low computational cost, the measures can be used as part of a manual, interactive search or combined with more rigorous mathematical optimization techniques. In the `diagnostics` module you can find examples that use flow diagnostics in combination with adjoint methods to determine optimal well locations and set optimal injection and production rates as discussed in more detail in [184].

COMPUTER EXERCISES:

90. Repeat the experiment using fixed rates for the producers (and possibly also for the injector). Can you explain the differences that you observe?
91. Use the interactive diagnostic tool introduced in the next section to manually adjust the bottom-hole pressures (or alternatively the production rates) to see if you can improve the sweep even further.
92. Can you devise an automated strategy that uses flow diagnostics to search for optimal five-spot patterns?
93. Cell-averaged time-of-flight values computed by a finite-volume scheme can be quite inaccurate when interpreted pointwise, particularly for highly heterogeneous media. To investigate this, pick one of the fluvial layers from SPE 10 and use the `pollock` routine from the `streamline` module to compute time-of-flight values on a 10×10 subsample inside a few selected cells. Alternatively, you can use the finite-volume method on a refined grid.

13.4 Interactive flow diagnostics tools

The examples you have encountered so far in the book have mostly been self contained in the sense that we have either discussed the code lines necessary for the example, or given reference to complete MRST scripts that can be run in batch or cell mode to produce the figures and numerical results discussed in the text. In this section, we will deviate slightly from this rule. While the ideas behind most flow diagnostics techniques are relatively simple to describe and their computation is straightforward to implement, the real strength of these techniques lies in their visual appeal and the ability for rapid user interaction. Together, MATLAB and MRST provide a wide variety of powerful visualization routines that can be used to visualize input parameters and simulation results. As you saw in the previous section, the `diagnostics` module supplies additional tools for enhanced visualization. However, using a script-based approach to visualization means that you each time need to write extra code lines to manually set color map and view angle or display various additional information such as legends, colorbars, wells, and figure titles, etc. These extra code lines have mostly been omitted in our discussion in the previous section, but if you go in and examine the accompanying scripts, you will see that a large fraction of the code lines focus on improving the visual appearance of plots. This code is repetitive and should ideally not be exposed to the user of flow diagnostics. More important, however, a script-based approach gives a static view of the data and offers limited capabilities for user interaction apart from zooming, rotating, and moving the displayed data sets. Likewise, a new script must be written and executed each time we want to look a new plot that combines various types of diagnostic data, e.g., to visualize time-of-flight or petrophysical values within a given color region, use time-of-flight to threshold the color regions, etc.

To simplify the user interface to flow diagnostics, we have integrates most of the flow-diagnostics capabilities into a graphical tool that enables you to interact more directly with your data set

```
interactiveDiagnostics(G, rock, W);
```

The script uses the standard two-point incompressible flow solver for a single-phase fluid with density 1000 kg/m^3 and viscosity 1 cP to compute a representative flow field, which is then fed to the function `computeTOFandTracer()` to compute the basic flow diagnostics quantities discussed above. Once the computation of basic flow diagnostic quantities is complete, the graphical user interface is launched, see Figure 13.17: This consists of a *plotting window* showing the reservoir model and a *control window* that contains a set workflow tabs and menus that enable you to use flow diagnostics to explore volumetric connections, flow paths, and dynamic heterogeneity measures. The control window has three different tabs. The 'region selection' tab is devoted to displaying the various kinds of volumetric regions discussed in Section 13.1.1. The 'plots' tab lets you compute $F\text{-}\Phi$ diagram, Lorenz coefficient, and the well-

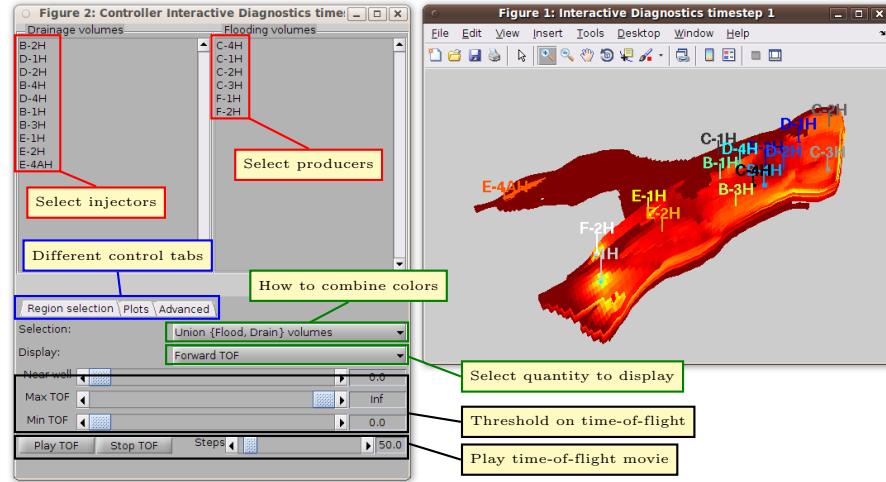


Fig. 13.17. The graphical user interface to flow diagnostics. The plotting window shows forward time-of-flight, which is the default value displayed upon startup. In the control window, we show the 'region selection' tab that lets you select which quantity to show in the plotting window, select which wells to include in the plot, specify how to combine color concentrations to select volumetric regions, and as well as set maximum and minimum time-of-flight values to crop the volumetric regions.

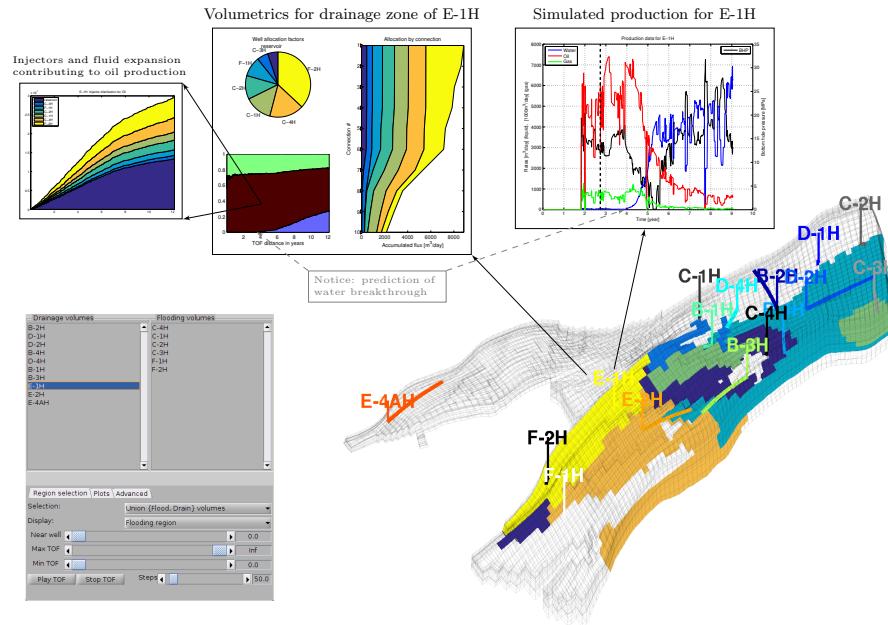


Fig. 13.18. Flow diagnostics used to post-process a three-phase, black-oil flow simulation of the Norne field.

allocation panels shown in Figure 13.11. From this tab, you can also bring up a dialog box that enables you to edit the well settings and recompute a new flow field and the resulting flow diagnostics (unless the parameter '`computeFlux`' is set to false). In the 'advanced' tab, you can control the appearance of the 3D plot by selecting whether to display grid lines, well-pair information, and well paths, set 3D lighting and transparency value, etc. This tab also allows you to export the current volumetric subset as a set of boolean cell indicators.

The regions to be displayed in the plotting window are specified by selecting a set of active wells and by choosing how the corresponding colorings should be combined. That is, for a given set of 'active' wells, you can either display all cells that have nonzero concentration for the injector *or* producer colors, only those cells that have nonzero concentration for injector *and* producer colors, only cells with nonzero concentration of producer colors, or only cells with nonzero concentrations for injector colors. To select a set of active wells, you can either use the list of injectors and producers in the control window, or you can select an individual well by left-clicking on the well in the plotting window. In the latter case, the set of active wells will consist of the well you selected plus all other wells that this well is in communication with. Clicking on a well will also bring up a new window that displays a pie chart of the well allocation factors and a graph that displays cumulative allocation factors per connection. Figure 13.18 shows this type of visualization for a model where we also have access to the various time steps of a full multiphase simulation. In this case, we invoke the GUI by calling

```
interactiveDiagnostics(G, rock, W, 'state', state, 'computeFlux', false);
```

so that the fluxes used to compute time-of-flight and color concentrations are extracted from the given reservoir state given in `state`. Left-click on producer E-1H, brings up a plot of the well allocation as well as a plot of the fluid distribution displayed as function of the backward time-of-flight from the well. To further investigate the flow mechanism, we can click on each fluid and get a plot of how the various injectors and fluid expansion in the reservoir contribute to push the given fluid toward the producer. In Figure 13.18 we have expanded the GUI by another function that enables us to plot the simulated well history of each well. The GUI also offers functionality to load additional cell-based data sets that can be displayed in the 3D plot:

```
interactiveDiagnostics(G, rock, W, celldata);
interactiveDiagnostics(G, rock, W, celldata, 'state', state);
```

The GUI also has more functionality for post-processing simulations with multiple time steps, but this is beyond the scope of the current presentation.

In the following we will use the interactive GUI to study a few reservoir models. This means that scripts accompanying the case studies in this section do not reproduce all figures directly as was the case for the script-based approach presented in the previous section. Instead, you will have to perform several manual actions specified in the scripts to reproduce some of the figures.

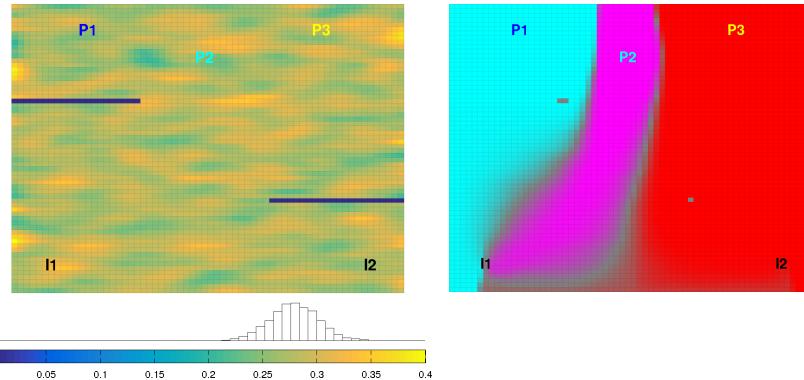


Fig. 13.19. A simple 2D reservoir with two injectors and three producers. The left plot shows porosity and the right plot the corresponding producer partition.

13.4.1 Simple 2D example

As our first example, we will use a slightly modified version of the setup used from Figure 13.1, in which we have introduced two low-permeable zones that will play the role of sealing faults, moved the two injectors slightly to the south, and switched all wells from rate to pressure control. The resulting reservoir model is shown in Figure 13.19, whereas specific values for the well controls are found in Table 13.2 under the label 'base case'. In our previous setup, we had a relatively symmetric well pattern in which producers P1 and P3 were supported by injectors I1 and I2, respectively, while producer P2 was supported by both injectors. This symmetry is broken by the two sealing faults, and now injector I1 also provides a significant support for producer P3. This can be inferred from the plot of the producer partition: the gray area between the magenta (P2) and red (P3) regions signifies a parts of the reservoir that are drained by both producers. Since there is a relatively large gray area southeast of I1, this injector will support both P2 and P3. There is also a gray area southwest of injector I2, but since this is less pronounced, we should expect that only a small portion of the inflow of P2 can be attributed to I2. To confirm this, you can load the model in the interactive viewer (see the script `interactiveSimple`), and click on the names for each individual producer to bring up a pie chart reporting the corresponding flux allocation.

Next, we let us try to figure out to what extent this is a good well pattern or not. We can start by looking at how a displacement front would propagate if the present flow field remains constant. If the displacement front travels with a unit speed relatively to the Darcy velocity given by the flux field, the region swept by the front at time t consists of all those cell for which $\tau_f \leq t$. Using the interactive GUI as shown in Figure 13.17, you can either show swept regions by specifying threshold values manually, or use the 'Play TOF' button to play a 'movie' of how the displacement front advances through the

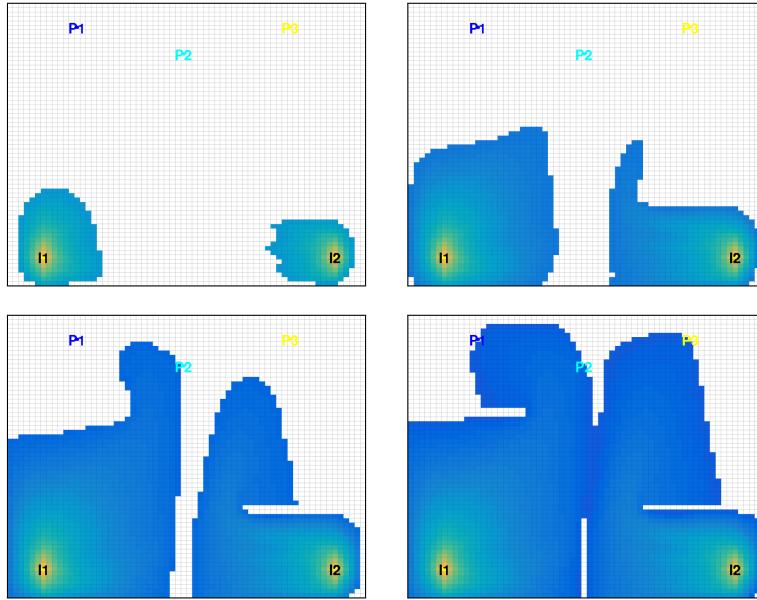


Fig. 13.20. Evolution of an imaginary displacement front illustrated by thresholding time-of-flight inside the sweep regions for the base case.

reservoir. Figure 13.20 shows four snapshots of such an advancing front. We notice, in particular, how the sealing fault to the northwest of the reservoir impedes the northbound propagation of the I1 displacement front and leads to early breakthrough in P2. We also see that there is a relatively large region that is still unswept after the two displacement fronts have broken through in all three producers.

As a more direct alternative to studying snapshots of an imaginary displacement front, we can plot the residence time, i.e., the sum of the forward and backward time-of-flight, as shown in the upper-left plot in Figure 13.21. Here, we have used a nonlinear gray-map to more clearly distinguish high-flow zones (dark gray) from stagnant regions (white) and other regions of low flow (light gray). In the figure, we see that wells I1 and P2 are connected by a high-flow region, which explains the early breakthrough we observed in Figure 13.20. The existence of high-flow regions can also be seen from the $F\text{-}\Phi$ diagram and the Lorenz coefficient of 0.273.

The interactive diagnostic tool has functionality that lets you modify the well controls and if needed, add new wells or remove existing ones. We will now use this functionality to try to improve the volumetric sweep of the reservoir, much in the same way as we did manually for a layer of SPE 10 in Section 13.3.2. We start by reducing the high flow rate in the region influenced by I1 and P2. That is, we increase the pressure in P2 to, say, 130 bar to decrease the inter-well pressure drop. The resulting setup is referred to

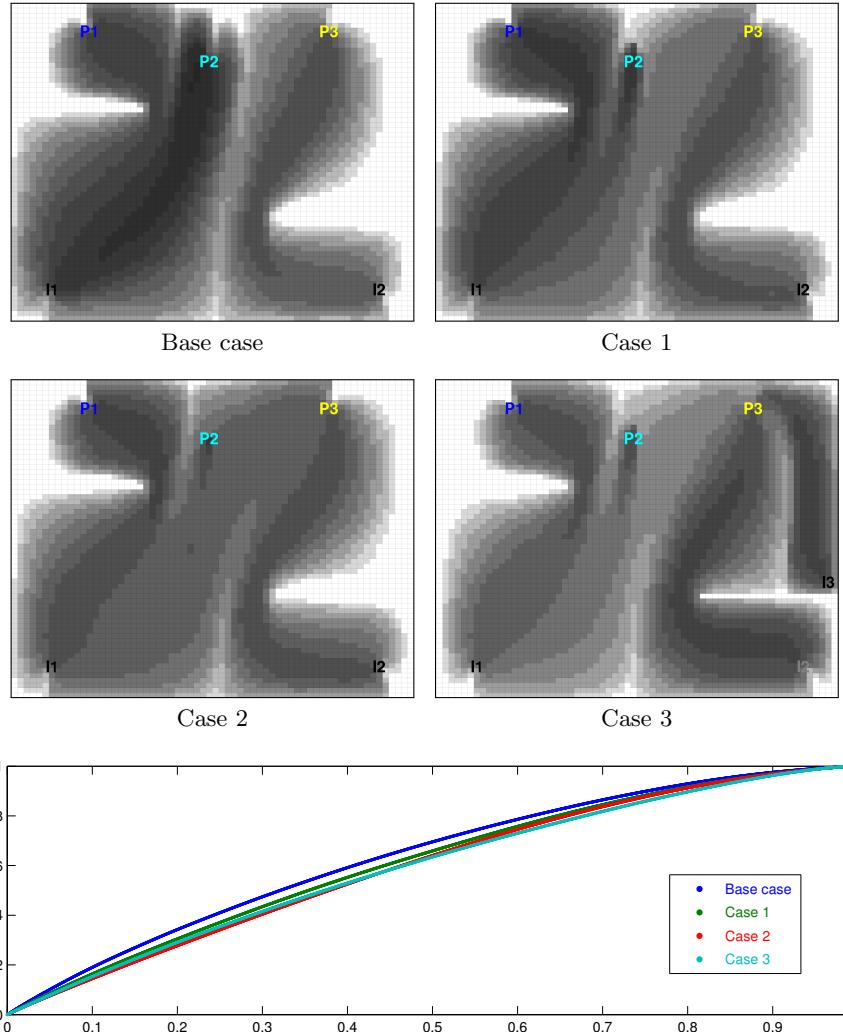


Fig. 13.21. In Cases 1 and 2, well controls have been manually adjusted from that of the base case (see Table 13.2) to equilibrate total travel time throughout the reservoir. Case 3 includes infill drilling of an additional injector. The bottom plot shows the corresponding F - Φ curves.

Table 13.2. Well controls given in terms of bottom-hole pressure [bar] for a simple 2D reservoir with five initial wells (I1, I2, P1, P2, P3) and one infill well (I3).

	I1	I2	I3	P1	P2	P3	Lorenz
Base case	200	200	—	100	100	100	0.2730
Case 1	200	200	—	100	130	80	0.2234
Case 2	200	200	—	100	130	80	0.1934
Case 3	200	220	140	100	130	80	0.1887

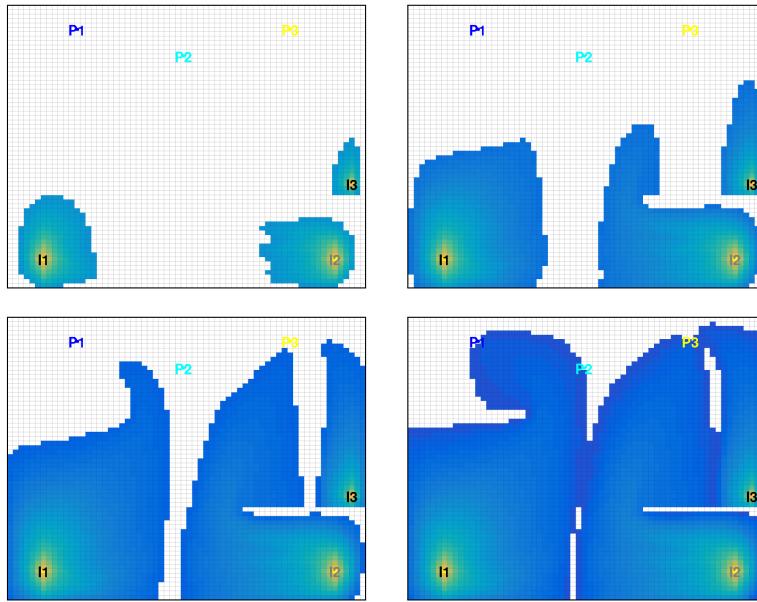


Fig. 13.22. Evolution of an imaginary displacement front illustrated by thresholding time-of-flight inside the sweep regions for Case 3.

as 'Case 1', and gives more equilibrated flow paths, as can be seen from the Lorenz coefficient and the upper-right plot in Figure 13.21. In Case 2, we have decreased the pressure in P3 to 80 bar to increase the flow in the I2–P3 region. As a result of these two adjustments to the well pressures, we have reduced the stagnant region north of P2 and also diminished the clear flow-divide that extended from the south of the reservoir to the region between P2 and P3. To also sweep the large unswept region east of P3, we can use infill drilling to introduce a new well in the southeast of this region, just north of the sealing fault. Since the new well is quite close to the existing producer, we should assign a relatively low pressure to avoid introducing too high flow rates. In Case 3, we have chosen to let the well operate at 140 bar and at the same time we have increased the pressure in I2 to 220 bar. Figure 13.22 shows snapshots of the advancing front at the same instances in time as was used in Figure 13.20 for the base case. Altogether, the well configuration of Case 3 gives a significant increase in the swept areas and reduces the Lorenz coefficient to 0.189. We can therefore expect this configuration to give better displacement if the setups were rerun with a multiphase simulator.

A more detailed description of how you should use the interactive GUI to perform the above experiments can be found in the `interactiveSimple.m` script of the `book` module. I encourage you to use the script to familiarize yourself with interactive flow diagnostics. Are you able to make further improvements?

13.4.2 SAIGUP: flow patterns and volumetric connections

The previous example was highly simplified and chosen mainly to illustrate the possibilities that lie in the interactive use of flow diagnostics. In this example, we revisit the SAIGUP case study from Section 5.4.4 on page 176 and take a closer look at the volumetric connection in this shallow-marine reservoir. We start by re-running this case study to set up the simulation model and compute a flow field:

```
saigupWithWells; close all
clearvars -except G rock W state
```

Henceforth, we only need the geological model, the description of the wells, and the reservoir state and have hence cleared all other variables and closed all plots produced by the script. With this, we are ready to launch the interactive flow diagnostics session. Since we already have computed a reservoir state, we can pass this on to the interactive GUI and hence use it in pure post-processing mode:

```
interactiveDiagnostics(G, rock, W, 'state', state, 'computeFlux', false);
```

in which we are not able to edit any well definitions and recompute fluxes.

In Figure 5.11 on page 177 we saw that although the injectors and producers are completed in all the twenty grid layers of the model, there is almost no flow in the bottom half of the reservoir. A more careful inspection shows that there is almost no flow in the upper layers in most of the reservoir either. This is a result of the fact that the best sand quality is found in the upper-middle layers of the reservoir, as shown in Figure 13.23, which compares the permeability in the full model with the permeability in cells having a residence time less than one hundred years. Because each injector is controlled by a total fluid rate, large fluid volumes will be injected in completions that are connected to good quality sand, while almost no fluid is injected into zones with low permeability and porosity. This can be seen in Figure 13.24, which shows overall and cumulative well-allocation factors for four of the injectors. Injectors I3 and I4 are completed in the southern part of the reservoir, and here low-quality sand in the bottom half of the reservoir leads to almost negligible injection rates in completions 11 to 20. In a real depletion plan, the injectors would probably not have been completed in the lower part of the sand column. Injector I5 located to the west in the reservoir is completed in a column with low permeability in the top four and the bottom layer, high permeability in Layers 6 to 9, and intermediate permeability in the remaining layers. Hence, almost no volume is injected through the top four completions, which hence are redundant. Finally, injector I6 is completed in a column with poor sand quality in the top three layers, high permeability in Layers 4 to 9, and intermediate permeability in the remaining layers.

Figure 13.24 also shows the flux allocation for all well pairs in the reservoir. In the plot, each curved line corresponds to a connection between an injector

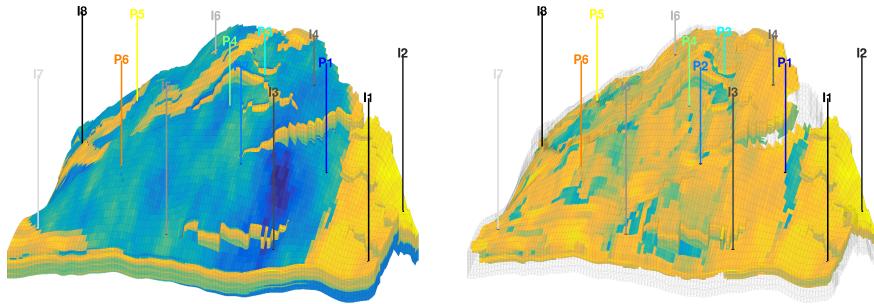


Fig. 13.23. Horizontal permeability ($\log_{10} K_x$) for the SAIGUP model. The left plot shows the full permeability field, while the right plot only shows the permeability in cells that have a total residence time less than 100 years. (The reservoir is plotted so that the north-south axis goes from left to right in the figure.)

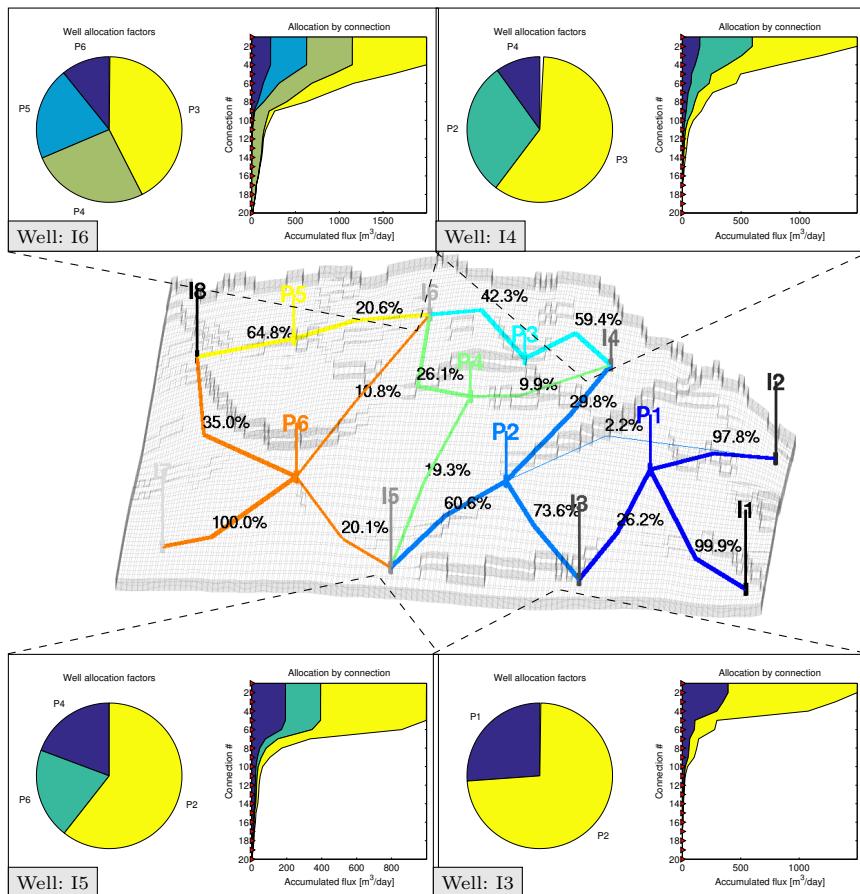


Fig. 13.24. Flux allocation for all well pairs of the SAIGUP model and well-allocation factors for injectors I3, I4, I5 and I6.

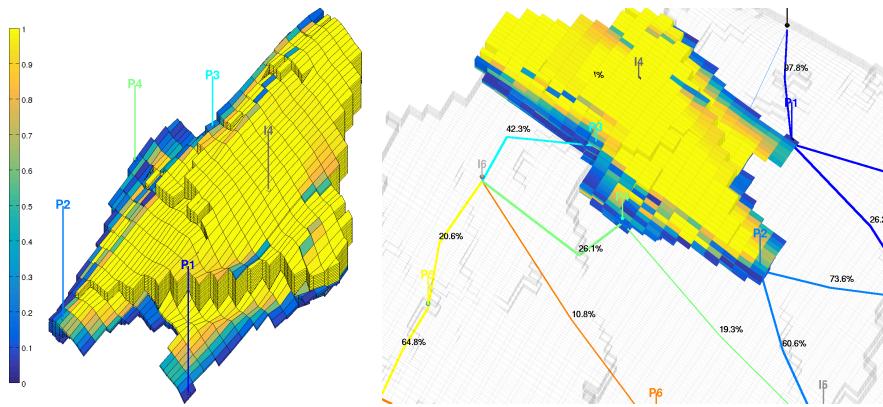


Fig. 13.25. Plot of the injector color region for well I4.

and a producer, where the color of the line signifies the producer and the percentage signifies the fraction of the total flux from each injector that goes to the different producers. The connections have been truncated so that only pairs that correspond to at least 1% of the flux are shown in the figure; which explains why not all the fractions sum up to unity. Let us take injector I4 as an example. Figure 13.25 shows two plots of the color region for this injector. From the well-allocation plots, we have already seen that I4 is connected to producers P2 to P4; (almost) all the well completions lie inside of the region colored by I4. Producer P1, on the other hand, is only completed in a single cell inside the colored region, and this cell is in the top layer of the reservoir where the sand quality is very poor. It is therefore not clear whether P1 is actually connected to I4 or if this weak connection is a result of inaccuracies in the computation of color concentrations. Since we only use a first-order discretization, the color concentrations will generally contain a significant amount of numerical smearing near flow divides, which here is signified by blue-green colors.

COMPUTER EXERCISES:

94. Change all the injectors to operate at a fixed bottom-hole pressure of 300 bar. Does this significantly change the flow pattern in the reservoir? Which configuration do you think is best?
95. Use the flow diagnostic tool to determine well completions in the SAIGUP model that have insignificant flow rate, eliminate these well completions, and rerun the model. Are there any apparent changes in the flux allocation and volumetric connections?
96. Consider the model given in `makeAnticlineModel.m`. Can you use flow diagnostics to suggest a better well configuration?

Grid Coarsening

Over the last decades, methods for mapping and characterizing subsurface rock formations have improved tremendously. This has, together with a dramatic increase in computational power, enabled the industry to build increasingly detailed and complex models to account for heterogeneous structures on different spatial scales. Using gridding techniques similar to the ones outlined in Chapter 3, one can today easily build complex geological models consisting of multiple millions of 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.

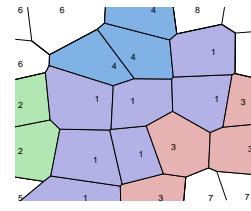
Through parallelization and use of high-performance computers it is possible to simulate fluid flow on grid models with millions of cells, at least in principle. In many practical engineering workflows, however, such large-scale computations may not be feasible. The user may not have access to parallel 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 methods will be discussed in more detail in Chapter 15.

Coarse grids can obviously be generated as discussed in Chapter 3 by simply specifying a lower 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 `agglo` module offers tools for defining flexible coarse grids that adapt to geological features and flow patterns, e.g., as discussed in [102, 101, 189, 158]. Coarse partitions also form a basis for contemporary multiscale methods and are used extensively in the various multiscale modules of MRST (`msmfem`, `msfv`, etc). In 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 14.2. 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.

14.1 Partition vectors

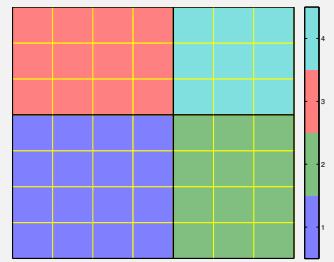
To demonstrate the simplicity and power of using a partition vectors to define coarse grids, we will go through a set of examples. Once you get the idea, it should be straightforward to use your own creativity for defining new partitions. (Complete codes for all examples discussed in this section are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

14.1.1 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([1.5 max(p)+.5]);
colormap(lines(max(p)));
set(colorbar, 'YTick', 1:max(p));
```

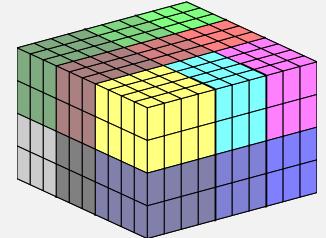


The call to `partitionUI` returns a vector with one element per cell taking one of the integer values 1, 2, 3, 4 that represent the four blocks. Since seven is not divisible by two, the coarse blocks will not have the same size: we will have blocks of size 4×4 , 3×3 , 4×3 , and 3×4 . To better distinguish 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
```



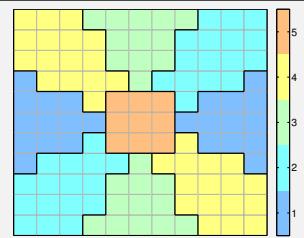
Here, we have used the `colorcube` colormap, which is particularly useful for visualizing partition vectors since it contains as many regularly spaced colors in RGB color space as possible. The careful reader will also observe that the arguments to `outlineCoarseGrid` changes somewhat for 3D grids.

14.1.2 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 530. 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:

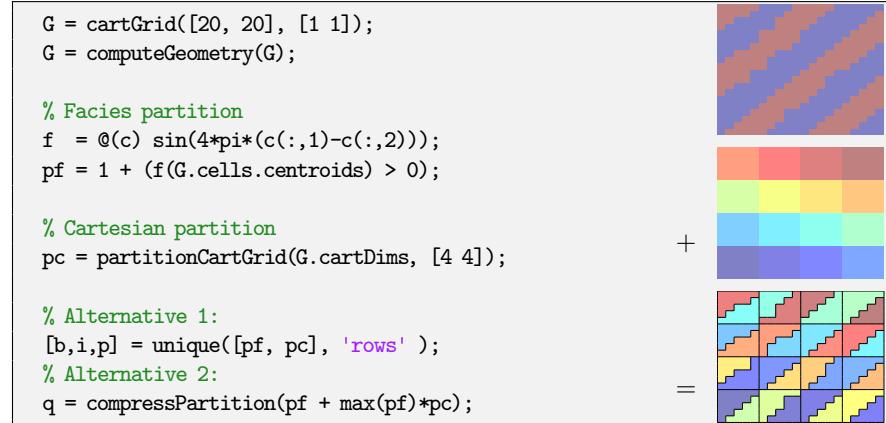
```
q = processPartition(G, p, facelist)
```

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

14.1.3 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.



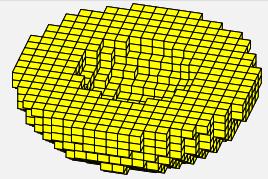
The example also shows two alternative techniques for combining different partitions. In the first alternative, we collect the partitions as columns in an array `A`. The call `[b,i,p]=unique(A,'rows')` will return `b` as the unique rows of `A` so that `b=A(i)` and `A=b(p)`. Hence, `p` will be a partition 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 renames 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 one can 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 67.

```

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]);

```

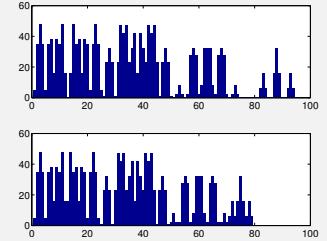


Figure 14.1 shows the partition obtained after we have compressed the partition vector. To clearly distinguish the different blocks, we have used an explosion view, which is a useful technique for visualizing coarse partitions. The complete code for this visualization method can be found in `explosionView.m`.

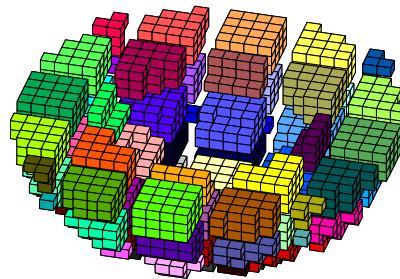


Fig. 14.1. The partition of the cup-formed grid visualized using an 'explosion view' type method.

14.2 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 connectivity 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

<code>CG.cells =</code>	<code>CG.faces =</code>
<code>num: 4</code>	<code>num: 12</code>
<code>facePos: [5x1 double]</code>	<code>neighbors: [12x2 double]</code>
<code>faces: [16x2 double]</code>	<code>connPos: [13x1 double]</code>
	<code>fconn: [24x1 double]</code>

Figure 14.2 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.

14.2.1 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

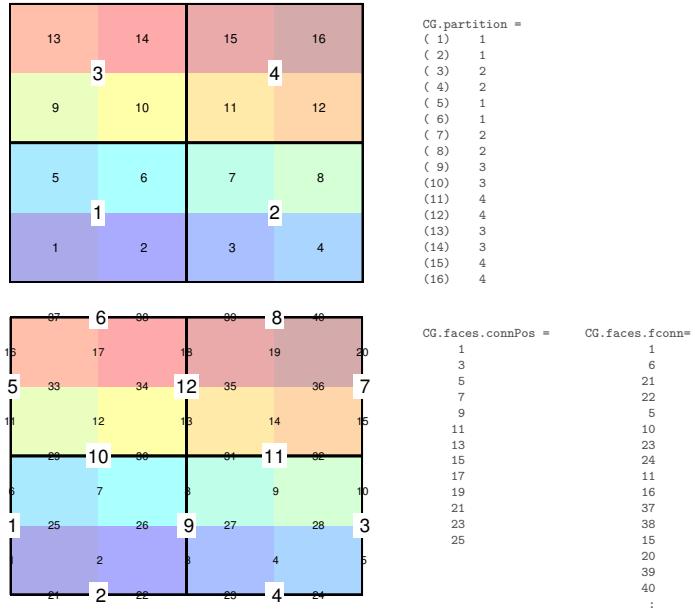


Fig. 14.2. Illustration of the relation between blocks in the coarse grid and cells in the parent grid (top) and between connections in the coarse grid and faces from the parent grid (bottom).

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 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 533 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, 'EdgeColor', 'none');
colormap((jet(16)+ones(16,3))/2); axis off;

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

CG = coarsenGeometry(CG);
plotFaces(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));

plotCellData(CG,(1:max(p))', 'EdgeColor', 'none');
plotFaces(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.

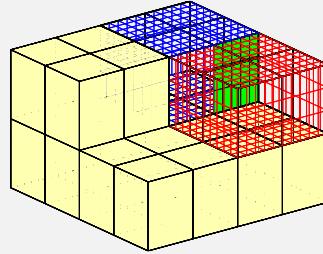
To demonstrate this, we will visualize one connection of a subdivided coarse face that consists of several fine faces, along with the fine cells that belong to the neighboring blocks:

```

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;
plotFaces(CG, show, 'FaceColor',[1 1 .7]);
plotFaces(G, ff, 'FaceColor', 'g')
plotFaces(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')

```



14.3 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 2.5.4 and the SAIGUP model from Section 2.5.5. The complete code for the two examples can be found in the scripts `coarsenJohansen.m` and `coarsenSAIGUP.m`.

14.3.1 The Johansen aquifer

As we saw in Section 2.5.4, 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 2.15, these three formations have distinctively different permeabilities. The model was originally developed to study a potential site for geological storage of CO₂ injected as a supercritical fluid deep in the formation. The injected CO₂ 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 CO₂ 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 CO₂ is to be kept, while the Dunlin shale acts like a seal that prevents the CO₂ to escape

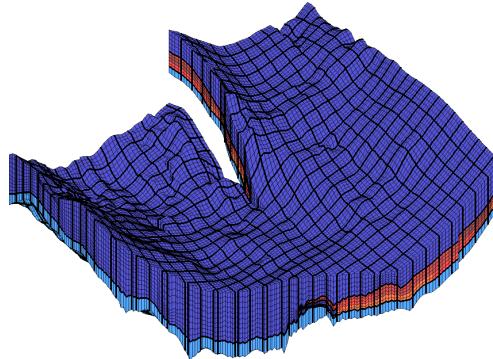


Fig. 14.3. A $4 \times 4 \times 11$ coarsening of the NPD5 model of the Johansen aquifer that preserves the Amundsen, Dunlin, and Johansen formations.

Table 14.1. Permeability values used to distinguish the different formations in the NPD5 sector model of the Johansen formation.

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

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 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 14.1

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 2.5.4, 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);

```

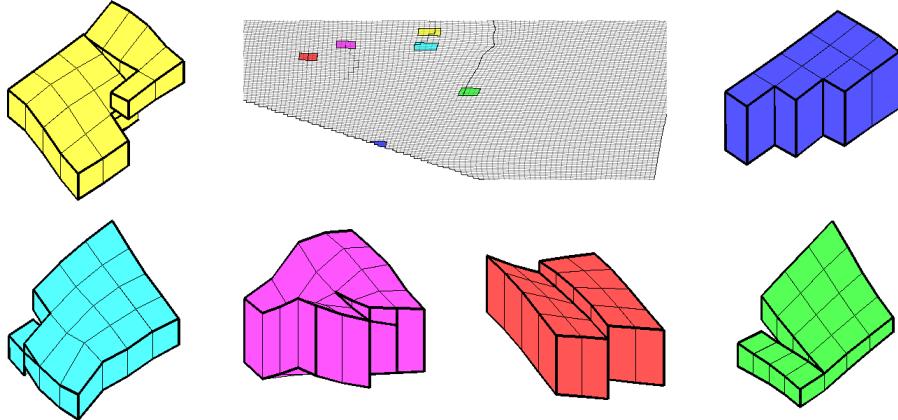


Fig. 14.4. Six coarse blocks sampled from the top grid layer of the Dunlin shale in a $4 \times 4 \times 1$ coarsening of the NPD5 sector model of the Johansen formation.

The resulting coarse grid is shown in Figure 14.3. By intersecting the partition vector p_C , which has only one block in the vertical direction, with the partition vector p_K that represents the different formations, we get three blocks in 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 than 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 14.4 shows 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 5. 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 15) 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.

14.3.2 The SAIGUP model

As our next example, we will revisit the SAIGUP model discussed on page 103. As shown in Figure 2.20 in Section 2.5.5, 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 14.5 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

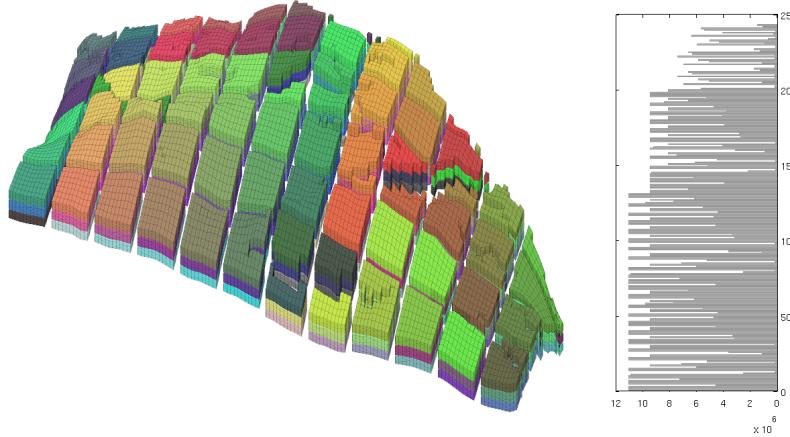


Fig. 14.5. Logically Cartesian partition of the SAIGUP model. The plot to the left shows the individual blocks in an explosion view using the `colrcube` colormap. The bar graph to the right shows the volumes in units [m^3] for each of the blocks in the partition.

of its neighbors. Then we update the partition vector by relabelling 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:

```

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 14.6 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 constraints 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 14.6. Here, the yellow and cyan blocks were created when partitioning the grid uniformly in index space. The basic `processPartition` routine only

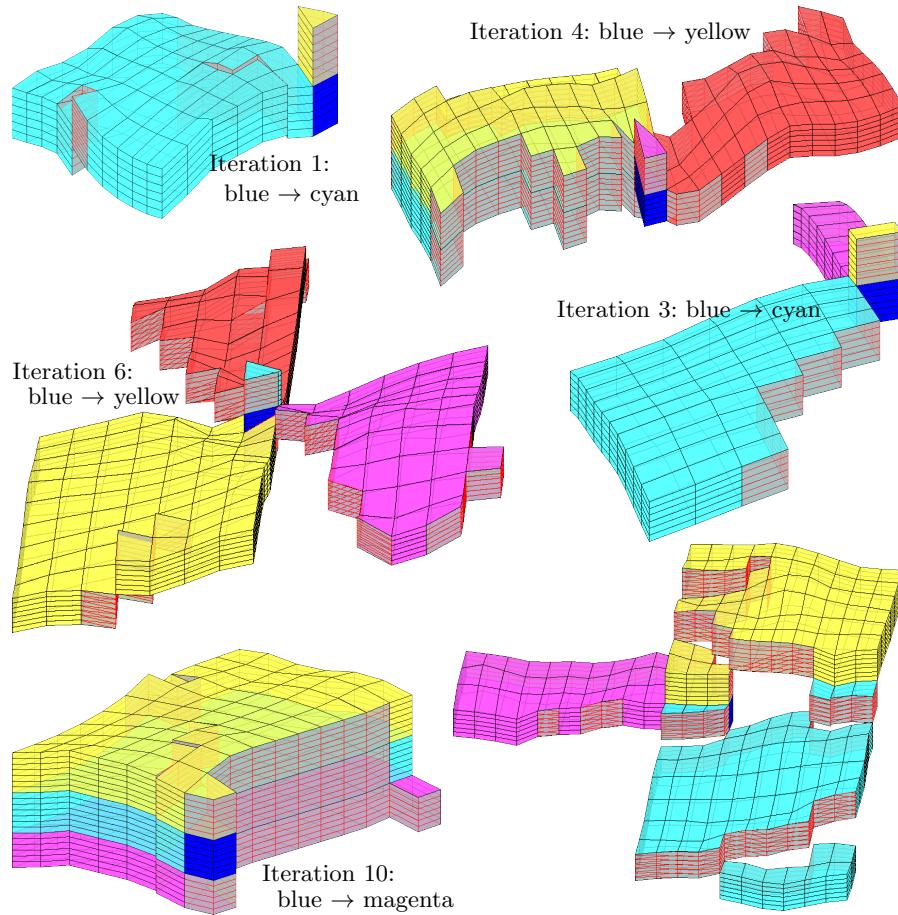


Fig. 14.6. Examples of small blocks (in blue) being merged with one of their neighbors shown in semi-transparent color with fault faces in gray.

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 areas when we postprocess the first uniform partition, e.g., do something like the following,

```
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, which is obtained 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 14.6. On the other hand, this approach will obviously involve a threshold parameter that will vary from case to case and will have 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.

14.4 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 (see [5]) 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 14.7 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 to enable inflow *and* outflow in the block.
 - 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.

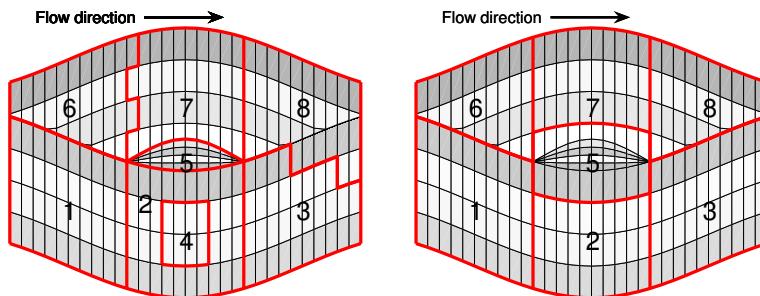


Fig. 14.7. Illustration of some of the guidelines for choosing a good coarse grid. In the left plot, all blocks except for Block 1 violate at least one of the guidelines each. In the right plot, the blocks have been improved at the expense of an increased number of connections.

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

In the right plot of Figure 14.7, 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 SPE 10 model in Section 2.5.3), 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 SPE 10, 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 2.20), 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 CO₂ 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.

15

Upscaling Petrophysical Properties

As you have seen previously in the book, porous rock formations are typically heterogeneous at a large variety of length scales, from the micrometer scale of pore channels to the kilometer scale of petroleum reservoirs or even larger scales for aquifer systems and sedimentary basins. Describing all flow processes pertinent to hydrocarbon production or CO₂ extraction with a single model is therefore not possible. In Section 2.3 we introduced you to the concept of representative elementary volumes and mentioned various types of models that can be used for flow studies, including: flow in core samples (cm scale), bed models (meter scale), sector models, and field models (km scale). These models must be calibrated against static and dynamic data of very different spatial (and temporal) resolution: thin sections, core samples, well logs, geological outcrops, seismic surveys, well tests, production data, core flooding and other laboratory experiments, etc.

For full sized reservoirs the traditional approach has been to model geological structures with a geological model, and fluid flow with a coarser simulation model. Geological models are produced to represent the heterogeneity of the reservoir and possibly incorporate a measure of inherent uncertainty. Pore, core, and bed models are mainly designed to give input to the geological characterization and to derive flow parameters for simulation models. The process of making a geological model is generally strongly under-determined. It is therefore customary, in particular on the reservoir scale, to use *geostatistical* methods to generate plausible distributions of petrophysical properties as discussed briefly in Section 2.5. In recent years, process models that seek to mimic deposition and subsequent structural changes have also gained a lot in popularity since they tend to produce less artifacts in the geo-cellular models. Heterogeneity and correlations in petrophysical properties depend strongly on the patterns in the sedimentary deposits and flow patterns tend to be strongly affected by details in the structural architecture of the reservoir. There is therefore a general trend to build complex, high-resolution models for geological characterization to represent small-scale geological structures. Likewise, large ensembles of equiprobable models are generated to systematically

quantify model uncertainty. While high-resolution models can describe a wide variety of geological structures, there are also many structures on a finer scale than the resolution of the geological model that are thought to be important to understand the reservoir. Therefore, it is common to develop hierarchies of models that cover a wide range of physical scales to systematically propagate the effects of small-scale geological variations observed in core samples up to the reservoir scale.

On the other hand, high-resolution geological used for reservoir characterization tend to contain more volumetric cells than contemporary simulators can handle without having to resort to massively parallel high-performance computing. As a result, flow simulations are usually performed using models that contain less details than those used for geological characterization. And even if you had all the computational power you needed, there are several arguments why you still should perform simulations on coarser models. First of all, one can argue that high-resolution models contain more details than what is justified by the available data. Second, in many modeling workflows the available computational power should be spent running multiple model realizations to span the range of plausible outcomes rather than on obtaining high numerical resolution for few highly uncertain predictions. Third, because coarser models contain fewer parameters, they are simpler to calibrate to observed reservoir responses (pressure tests and production data). Finally, a coarser model may be sufficient to predict flow patterns and reservoir responses with the accuracy needed to make a certain business decision. It is perhaps tempting to believe that with future increases in computing power, one will soon be able to close the gap in resolution between models used for reservoir characterization and models used for flow simulation. The development so far, indicates that such an idea is wrong. The trend is rather that increases in computational power enable geologists and reservoir engineers to build larger and more complex models at a pace that outperforms the improvement in simulation capabilities.

For all the reasons above, there is a strong need for mathematical and numerical techniques that can be used to communicate parameters and properties between models of different spatial resolution. And – this need will persist, and maybe even grow stronger, in the foreseeable future.

Upscaling (or homogenization) refers to the process of propagating properties and parameters from a model of *high spatial resolution* to a model of *lower spatial resolution*. In this process, *heterogeneous regions* in a reservoir model are replaced by *homogeneous regions* to make up a coarser model of the same reservoir. Or, in the words of Chapter 14 on grid coarsening, upscaling is the process in which petrophysical properties in the cells that make up a coarse block are averaged into a single effective value for each coarse block, see Figure 15.1. The effective properties of the new homogeneous regions are defined so that they preserve the effects of small-scale variations in an averaged sense. How this averaging should be performed depends on the type of property to be upscaled. One distinguishes between *additive properties* that

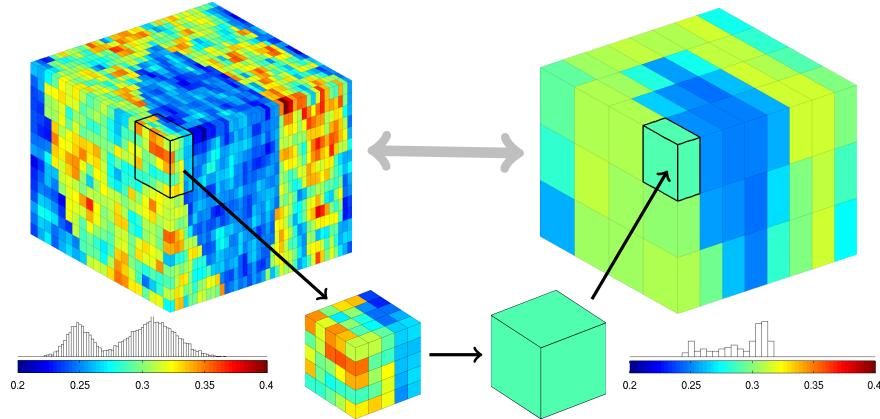


Fig. 15.1. Upscaling of petrophysical properties, here represented by porosity.

can be upscaled using simple volumetric averaging and *nonadditive properties*, for which correct averaging methods only exist in special cases and the best one can hope for in the general case is to compute accurate approximations.

Downscaling methods are used when propagating properties and parameters from a model with *low spatial resolution* to a model having *higher spatial resolution*. One can, for instance, be interested in refining coarse-scale modifications in petrophysical properties obtained during a history match on a simulation model to update the underlying geomodel. In this process, the aim is to preserve both the coarse-scale trends and the fine-scale heterogeneity structures.

15.1 Upscaling for reservoir simulation

The process of upscaling petrophysical parameters leads to many fundamental questions. For instance, do the partial differential equations that make up the flow model on the coarse scale take the same form as the equations modeling flow at the subgrid scale? And if so, how do we honor the fine-scale heterogeneities at the subgrid level. Even though upscaling has been a standard procedure in reservoir simulation for more than four decades, nobody has answered these questions rigorously, except for cases with special heterogeneous formations such as periodic or stratified media.

Homogenization is a rigorous mathematical theory for asymptotic analysis of periodic structures, see [37, 120, 110]. A relevant result states that for a periodic medium with permeability $K(\frac{x}{\varepsilon})$, there exists a constant symmetric and positive-definite tensor K_0 such that the solutions p_ε and $v_\varepsilon = -K(\frac{x}{\varepsilon})\nabla p_\varepsilon$ to the following elliptic problem

$$-\nabla \cdot K\left(\frac{x}{\varepsilon}\right)\nabla p_\varepsilon = q \quad (15.1)$$

converge uniformly as $\varepsilon \rightarrow 0$ to the corresponding solutions of a homogenized equation

$$-\nabla \cdot \mathbf{K}_0 \nabla p_0 = q. \quad (15.2)$$

This means that if we assume that the region we seek to compute an effective property for is a cell in an infinite periodic medium, then homogenization theory can be used to derive homogenized tensors for the region. The main advantage of this approach is that it provides mathematical methods to prove existence and uniqueness of the solution, and also verifies that the governing equation at the macroscopic level take the same form as the elliptic equation that governs porous media flow at the level of the representative elementary volumes (REVs). However, whether homogenization can be used for practical simulations is more debatable, in particular since natural rocks very seldom are periodic.

As you will see shortly, the most basic upscaling techniques rely on some kind of local averaging procedure in which effective properties in each grid block are calculated solely from properties within the grid block. As such, these averaging procedures do not consider coupling beyond the local domain, which in turn implies that the upscaling methods fail to account for the effect of long-range correlation and large-scale flow patterns in the reservoir unless these can be represented correctly by the forces that drive flow inside the local domain. Because different flow patterns may call for different upscaling procedures, it is generally acknowledged that global effects must also be taken into consideration to obtain robust coarse-scale simulation models.

Upscaling must also be seen in close connection with gridding and coarsening methods, as discussed in Chapters 3 and 14. Even though these questions have already been answered when making the fine-scale model, you should consider what kind of grid you want to use to represent the porous medium on the coarse scale, what resolution you need, and how you should orient your coarse blocks. To make the simulation models as accurate and robust as possible, the grid should be designed so that grid blocks capture heterogeneities on the scale of the block. This often implies that you may need significantly more blocks if you use a regular grid than if you use an unstructured polyhedral grid. You may also want to use different coarsening factors in zones of high and low flow, near and far away from wells and fluid contacts, etc. The importance of designing a good coarse grid should not be underestimated, but gridding in itself will not be discussed any further herein. Henceforth, we will instead tacitly assume that a suitable coarse grid is available and focus on upscaling techniques.

The literature on upscaling techniques is extensive, ranging from simple averaging techniques, e.g., [122], via flow-based methods that rely on local flow problems [33, 77] to more comprehensive global [192, 109] and local-global [54, 55, 93] methods. Some attempts have been made to analyze the upscaling process, e.g., [27, 262], but so far there is generally no theory or framework for assessing the quality of an upscaling technique. In fact, up-

scaling techniques are seldom rigorously quantified with mathematical error estimates. Instead, the quality of upscaling techniques is usually assessed by comparing upscaled production characteristics with those obtained from a reference solution computed on an underlying fine grid.

In the following, we will only discuss the basic principles and try to show you how you can implement relatively simple upscaling methods and explain why some methods may work well for some flow scenarios and not for others. If you are interested in a comprehensive overview, you should consult one of the many review papers devoted to this topic, e.g., [61, 256, 27, 222, 87, 78, 79]. We start by a brief discussion of how to upscale porosity and other additive properties, before we move on to discuss upscaling permeability effects, which is the primary example of a nonadditive property. Because of the way Darcy's law has been extended to from single-phase to multiphase flow, it is common to distinguish the upscaling of absolute permeability K , which is associated with single-phase flow, and relative permeability $k_{r\alpha}$, which is the multiphase factor that absolute permeability must be multiplied with to get the effective permeability experienced by one fluid in the presence of another. Upscaling of absolute permeability is therefore often called *single-phase upscaling*, whereas upscaling of relative permeability is referred to as *multiphase upscaling*. The main parts of this chapter is devoted to permeability upscaling and to upscaling of the corresponding transmissibilities that account for permeability effects in finite-volume discretizations. As in the rest of the book, our discussion will to a large extent be driven by examples, for which complete codes can be found in the `upscale` subdirectory of the `book` module of MRST.

15.2 Upscaling additive properties

Porosity is the simplest example of an additive property and can be upscaled through a simple volumetric average. If Ω denotes the region we want to average over, the averaged porosity value is given as

$$\phi^* = \frac{1}{\Omega} \int_{\Omega} \phi(\vec{x}) d\vec{x}. \quad (15.3)$$

Implementing the computation of this volumetric average can be a bit tricky if the coarse blocks and the fine cells are not matching. In MRST, however, we always assume that the coarse grid is given as a partition of the fine grid, as explained in Chapter 14. If q denotes the vector of integers describing the coarse partition, upscaling porosity amounts to a single statement

```
crock.poro = accumarray(q,rock.poro.*G.cells.volumes)./ ...
    max(accumarray(q,G.cells.volumes),eps);
```

Here we have used `max(..,eps)` to safeguard against division by zero in the unlikely case your grid contains blocks with zero volume or your partition

vector is not contiguous. Weighting by volume is not necessary for grids with uniform cell sizes. Similar statements were used to go from the fine-scale model on the left side of Figure 15.1 to the coarse-scale model shown on the right. Complete source code is found in the script called `illustrateUpscaling.m`. Flux densities are upscaled in the same way, with cell volumes replaced by face areas.

Other additive (or volumetric) properties like net-to-gross, saturations and concentrations can be upscaled almost in the same way, except that one now should replace the bulk average in (15.3) by a *weighted* average. If n denotes net-to-gross, the correct average would be to weight with porosity, so that

$$n^* = \left[\int_{\Omega} \phi(\vec{x}) d\vec{x} \right]^{-1} \int_{\Omega} \phi(\vec{x}) n(\vec{x}) d\vec{x}. \quad (15.4)$$

and likewise for saturations. In MRST, we can compute this upscaling as follows

```
pv = rock.poro.*G.cells.volumes;
N  = accumarray(q,pv.*n)./ max(accumarray(q,pv),eps);
```

To verify that this is the correct average, we simply compute

$$\begin{aligned} \phi^* n^* &= \left[\frac{1}{\Omega} \int_{\Omega} \phi(\vec{x}) d\vec{x} \right] \left[\int_{\Omega} \phi(\vec{x}) d\vec{x} \right]^{-1} \int_{\Omega} \phi(\vec{x}) n(\vec{x}) d\vec{x} \\ &= \frac{1}{\Omega} \int_{\Omega} \phi(\vec{x}) n(\vec{x}) d\vec{x} = (\phi n)^*. \end{aligned}$$

Using the same argument, one can easily argue that concentrations should be weighted with saturations, and so on. Rock type (or flow unit), on the other hand, is not an additive property, even though it is sometimes treated almost as if it was by applying a majority vote to identify the rock type that occupies the largest volume fraction of a block. Such a simple approach is generally not robust [266] and should generally be avoided.

To get more acquainted with upscaling of additive quantities, we recommend that you try to do the following computer exercises.

COMPUTER EXERCISES:

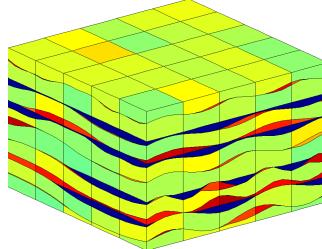
97. Construct a $8 \times 24 \times 5$ coarse partition of the SAIGUP model and upscale the additive rock properties. Verify your upscaling by computing

```
pv = accumarray(CG.partition, poreVolume(G,rock));
max(abs(pv - poreVolume(CG, crock)))
```

How would you make a coarse grid that represents the porosity better? To measure the quality of a given approximation, you can for instance use the following error measure:

```
err3 = sum(abs(rock.poro - crock3.poro(CG3.partition)))/...
sum(rock.poro)*100;
```

98. The file `mortarTestModel` from the `BedModels1` data set describes a sedimentary bed that consists of three different facies. Construct a coarse model that has 5×5 blocks in the lateral direction and as many blocks in the vertical direction as required to preserve distinct facies layers, as shown in the plot below.



Upscale the porosity of the model. Do you think the resulting coarse grid is suitable for flow simulations? (Hint: Check the guidelines from Section 14.4.)

15.3 Upscaling absolute permeability

To study upscaling of absolute permeability, it is sufficient to consider single-phase flow in the form of a variable-coefficient, Poisson equation

$$\nabla \cdot K \nabla p = 0. \quad (15.5)$$

Even with such a simple equation, the choice of what is the best method to average absolute permeability generally depends on a complex interplay between the local permeability distribution and the characteristic flow directions. In certain special cases, one can develop simple methods that average permeability correctly, but in the general case, all one can do is to develop computational methods that approximate the true effective permeability of the upscaled region. How accurate a given approximation is, will depend on the coarse grid, the specific upscaling method, the purpose for which the up-scaled values are to be used, and the complexity of the fine-scale permeability distribution.

Most techniques for upscaling absolute permeability seek an averaged tensor K^* that reproduces the same total flow through each homogeneous region as one would obtain if the single-pressure equation (15.5) was solved with the full fine-scale heterogeneity. In other words, if Ω is the homogeneous region to which we wish to assign an effective property K^* , this property should fulfill

$$\int_{\Omega} K(\vec{x}) \nabla p \, d\vec{x} = K^* \int_{\Omega} \nabla p \, d\vec{x}. \quad (15.6)$$

This equation states that the net flow rate \vec{v}_Ω through Ω is related to the average pressure gradient $\nabla_\Omega p$ in Ω through the upscaled Darcy law

$$\vec{v}_\Omega = -K^* \nabla_\Omega p. \quad (15.7)$$

The upscale permeability tensor K^* is not uniquely defined by (15.6) for a given pressure field p , and conversely, there does not exist a unique K^* so that (15.6) holds for any pressure field. This reflects that K^* depends on the flow through Ω , which in turn is determined by the boundary conditions that are specified on $\partial\Omega$. The better you know the boundary conditions the homogenized region will be subject to in subsequent simulations, the more accurate estimates you can compute for the upscaled tensor K^* . In fact, if you know these boundary conditions exactly, you can compute the true effective permeability. In general, you will not know these boundary conditions unless you have already solved your problem, and the best you can do is to make an educated and representative guess that aims to give reasonably accurate results for a wide range of flow scenarios. Another problem is that even though the permeability tensor of a physical system needs to be symmetric and positive definite (i.e., $\vec{z} \cdot K \vec{z} > 0$ for all nonzero \vec{z}), there is generally no guarantee that the effective permeability tensor constructed by an upscaling algorithm fulfill these properties. The possible absence of symmetry and positive definiteness shows that the single-phase upscaling problem is fundamentally ill-posed.

15.3.1 Averaging methods

The simplest way to upscale permeability is to use an analytic averaging formula. One general class of such formulas is the power average,

$$K^* = A_p(K) = \left(\frac{1}{|\Omega|} \int_\Omega K(\vec{x})^p d\vec{x} \right)^{1/p}. \quad (15.8)$$

Here, $p = 1$ and $p = -1$ correspond to the arithmetic and harmonic means, respectively, while the geometric mean is obtained in the limit $p \rightarrow 0$ as

$$K^* = A_0(K) = \exp \left(\frac{1}{|\Omega|} \int_\Omega \log(K(\vec{x})) d\vec{x} \right). \quad (15.9)$$

The use of power averaging can be motivated by the so-called Wiener-bounds [261], which state that for a statistically homogeneous medium, the correct upscaled permeability will be bounded above and below by the arithmetic and harmonic mean, respectively.

To motivate (15.8), we can look at the relatively simple problem of upscaling permeability within a one-dimensional domain $[0, L]$. From the flow equation (15.5) and Darcy's law, we have that

$$-(K(x)p'(x))' = 0 \implies v(x) = K(x)p'(x) \equiv \text{constant}$$

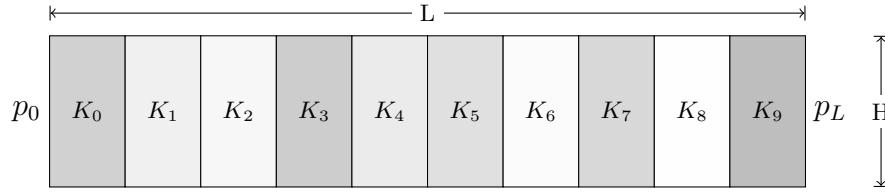


Fig. 15.2. Example of a perfectly stratified isotropic medium with layers perpendicular to the direction of the pressure drop, for which harmonic averaging is the correct way to upscale absolute permeability.

Since the upscaled permeability K^* must satisfy Darcy's law on the coarse scale, we have that $v = -K^*[p_L - p_0]/L$. Alternatively, this formula can be derived by using (15.6)

$$K^* \int_0^L p'(x)dx = K^*(p_L - p_0) \stackrel{(15.6)}{=} \int_0^L K(x)p'(x)dx = - \int_0^L v dx = -Lv.$$

We can now use (15.6) and Darcy's law to find the expression for K^*

$$\begin{aligned} \int_0^L p'(x)dx &= - \int_0^L \frac{v}{K(x)} dx \\ &= K^* \frac{p_L - p_0}{L} \int_0^L \frac{1}{K(x)} dx = K^* \left(\int_0^L p'(x)dx \right) \left(\frac{1}{L} \int_0^L \frac{1}{K(x)} dx \right), \end{aligned}$$

from which it follows that the correct way to upscale K is to use the harmonic average

$$K^* = \left(\frac{1}{L} \int_0^L \frac{1}{K(x)} dx \right)^{-1}. \quad (15.10)$$

This result is generally valid only in one dimension, but also applies to the special case of a perfectly stratified isotropic medium with layers perpendicular to the direction of pressure drop as illustrated in Figure 15.2. It is a straightforward exercise to extend the analysis above to prove that harmonic averaging is the correct upscaling in this case. (If you are familiar with Ohm's law in electricity, you probably realize that this setup is similar to that of resistors set in parallel.)

Computing the harmonic average of a permeability field is straightforward in MRST:

```

vol = G.cells.volumes;
for i=1:size(rock.perm,2)
    crock.perm(:,i) = accumarray(q,vol) ./ ...
                    accumarray(q,vol./rock.perm(:,i))
end

```

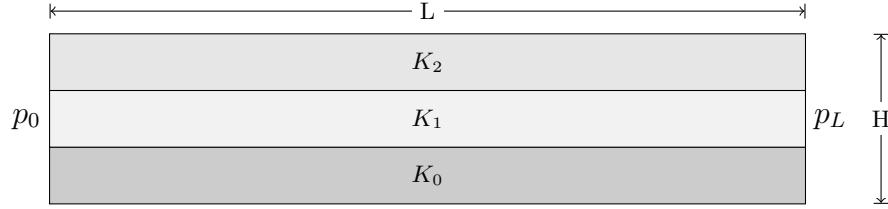


Fig. 15.3. Example of a perfectly stratified isotropic medium with layers parallel to the direction of the pressure drop, for which arithmetic averaging is the correct way to upscale absolute permeability.

where we, as in the previous section, have assumed that \mathbf{q} is a partition vector that describes the coarse grid.

Another special case is when the layers of a perfectly stratified isotropic medium are parallel to the pressure drop as shown in Figure 15.3. Since the permeability is constant along the direction of pressure drop, the pressure will be linear in the x -direction with $p(x, y) = p_0 + x(p_L - p_0)/L$. Using (15.6), we can compute

$$\begin{aligned} K^* \int_0^H \int_0^L \partial_x p(x, y) dx dy &= K^* H (p_L - p_0) \\ &= \int_0^H \int_0^L K(x, y) \partial_x p(x, y) dx dy = \frac{p_L - p_0}{L} \int_0^H \int_0^L K(x, y) dx dy, \end{aligned}$$

from which it follows that the correct upscaling is to use the arithmetic average

$$K^* = \frac{1}{LH} \int_0^H \int_0^L K(x, y) dx dy. \quad (15.11)$$

These examples show that averaging techniques can give correct upscaling in special cases, also in three dimensional space. If we now combine these two examples, we see that we can define the following upscaled permeability tensors for the isotropic media shown in Figures 15.2 and 15.3,

$$\mathbf{K}^* = \begin{bmatrix} A_{-1}^x(K) & 0 \\ 0 & A_1^x(K) \end{bmatrix} \quad \text{and} \quad \mathbf{K}^* = \begin{bmatrix} A_1^y(K) & 0 \\ 0 & A_{-1}^y(K) \end{bmatrix},$$

where the superscripts x and y on the averaging operator A from (15.8) signifies that the operator is only applied in the corresponding spatial direction. These averaged permeabilities would produce the correct net flow across the domain when these two models are subject to a pressure differential between the left and right boundaries or between the top and bottom boundaries. For other boundary conditions, however, the upscaled permeabilities will generally give incorrect flow rates.

To try to model flow in more than one direction, also for cases with less idealized heterogeneous structures modelled by a diagonal fine-scale tensor,

one can generate a diagonal permeability tensor with the following diagonal components

$$\mathbf{K}^* = \begin{bmatrix} A_1^{yz}(A_{-1}^x(\mathbf{K})) & 0 & 0 \\ 0 & A_1^{xz}(A_{-1}^y(\mathbf{K})) & 0 \\ 0 & 0 & A_1^{xy}(A_{-1}^z(\mathbf{K})) \end{bmatrix}.$$

In other words, we start by computing the harmonic average of each of the diagonal permeability components in the corresponding longitudinal direction, i.e., compute the harmonic average of K_{xx} along the x direction, and so on. Then, we compute the arithmetic average in the transverse directions, i.e., in the y and z directions for K_{xx} , and so on. This average is sometimes called the harmonic-arithmetic average and may give reasonable upscaling for layered reservoirs when the primary direction of flow is along the layers. More important, the harmonic-arithmetic average provides a tight lower bound on the effective permeability whereas the opposite method, the arithmetic-harmonic average, provides a tight upper bound.

While it is not obvious how to compute the harmonic-arithmetic average for a general unstructured grid, it is almost straightforward to use MRST to this end for rectilinear and corner-point grids. For brevity, we only show the details for the case when the grid has been partitioned uniformly in index space:

```

q = partitionUI(G, coarse);
vol = G.cells.volumes;
for i=1:size(rock.perm,2)
    dims = G.cartDims; dims(i)=coarse(i);
    qq = partitionUI(G, dims);
    K = accumarray(qq,vol)./accumarray(qq,vol./rock.perm(:,i));
    crock.perm(:,i) = accumarray(q,K(qq).*vol)./accumarray(q,vol);
end

```

The key idea in the implementation above is that to compute the harmonic average in one axial direction, we introduce a temporary partition `qq` that coincides with the coarse grid along the given axial direction and with the original fine grid in the other axial directions. This way, the call to `accumarray` has the effect that the harmonic average is computed for one longitudinal stack of cells at the time inside each coarse block. To compute the arithmetic average, we simply map the averaged values back onto the fine grid and use `accumarray` over the original partition `q`.

The script `averagingExample1` in the `book` module shows an example of the averaging methods discussed above. The permeability field is a 45×60 sub-sample of Layer 46 from the fluvial Upper Ness formation in the SPE 10 data set. In Figure 15.4, we have compared the effective permeabilities on a 15×15 coarse grid computed by arithmetic, harmonic, and harmonic-arithmetic averaging. From the plots, we see that arithmetic averaging has a tendency to

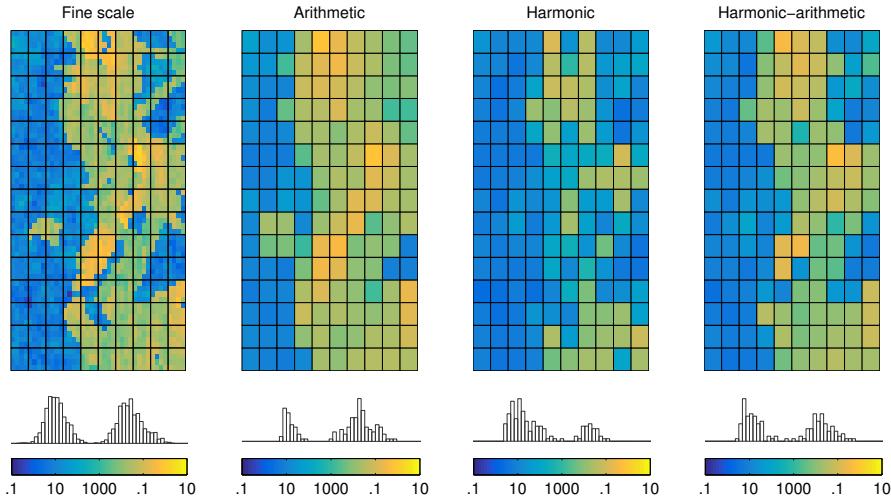


Fig. 15.4. Arithmetic, harmonic, and harmonic-arithmetic averaging applied to a 40×60 subset of Layer 46 from the SPE 10 data set.

preserve high permeability values, harmonic averaging tends to preserve small permeabilities, while harmonic-arithmetic averaging is somewhere in between.

Although simple averaging techniques like the ones discussed above can give correct upscaling in special cases, they tend to perform poorly in practice because the averages do not properly reflect the heterogeneity structures. Likewise, it is also generally difficult to determine which averaging to use since what is the best averaging method depends both on the heterogeneity of the reservoir and the prevailing flow directions. To illustrate this, we consider $8 \times 8 \times 8$ reservoir with three different permeability realizations shown in Figure 15.5. The three realizations are upscaled to a single coarse block using arithmetic, harmonic, and harmonic-arithmetic averaging. To assess the quality of the upscaling, we compare fine-scale and coarse-scale prediction of net flux across the outflow boundary for three different flow patterns: from west to east, from south to north, and from bottom to top. Complete source code for this simulation setup can be found in the script `averagingExample2` in the `book` module.

Table 15.1 reports the ratio between the outflow computed by the coarse models and the outflow computed on the original fine grid. For the layered model, arithmetic and harmonic-arithmetic averaging correctly reproduces flow in the lateral directions, whereas flow normal to the layers is correctly reproduced by harmonic and harmonic-arithmetic averaging. For the two anisotropic models from SPE 10, on the other hand, the flow rates predicted by the arithmetic and harmonic methods are generally far off. The combined harmonic-arithmetic method is more accurate, with less than 15% discrepancy for the Tarbert model and 17–76% discrepancy for the Upper

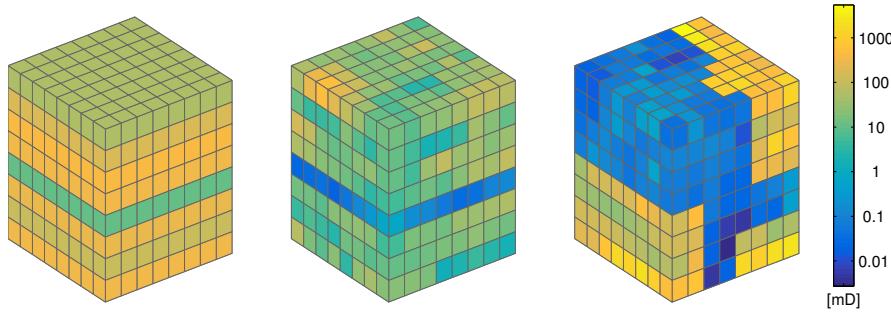


Fig. 15.5. Three models used to test the accuracy of averaging techniques. From left to right: a layered model and two subsets from the Tarbert and the Upper Ness formation, respectively, from the SPE 10 data set.

Table 15.1. Ratio between flow rate predicted by upscaled and fine-scale models for three different permeability fields, three flow scenarios, and three upscaling methods.

Model	Flow pattern	Arithmetic	Harmonic	Harm-arith
Layered	East→West	1.0000	0.2662	1.0000
	North→South	1.0000	0.2662	1.0000
	Top→Bottom	3.7573	1.0000	1.0000
Tarbert	East→West	1.6591	0.0246	0.8520
	North→South	1.6337	0.0243	0.8525
	Top→Bottom	47428.0684	0.3239	0.8588
Upper Ness	East→West	3.4060	0.0009	0.8303
	North→South	1.9537	0.0005	0.7128
	Top→Bottom	6776.8493	0.0020	0.3400

Ness model. Whether this can be considered an acceptable result will depend on the what purpose the simulation is to be used for.

COMPUTER EXERCISES:

99. Set up a set of flow simulations with different boundary conditions and/or well patterns to test the accuracy of the three effective permeability fields computed in `averagingExample1`.
100. Implement harmonic-arithmetic averaging for the SAIGUP model. (Hint: the script `cpGridHarmonic` in the `upscale` module computes harmonic averaging for the SAIGUP model).
101. Implement the operator A_p in (15.8) as a new utility function.

15.3.2 Flow-based upscaling

In our argumentation above, we repeatedly used the simple flow problems to argue whether a particular averaging method was good or not. Taking this idea one step further, we could impose representative boundary conditions along the perimeter of each coarse block and solve the flow problem (15.5) numerically to determine fine-scale pressures and flow rates and use these to define net flow rates and average pressure gradients, which can be inverted to determine effective coarse-scale permeabilities from Darcy's law. That is, once p and v have been computed, we use the same type of argument as we used to determine the formulas for the harmonic and arithmetic averages, except that we now work with discretely defined pressures and fluxes. This raises the immediate question: what kind of boundary conditions should we use?

A number of different boundary conditions have been suggested over the years. One obvious alternative is to use three different sets of boundary conditions for each block to create a pressure drop across the coarse cell in each of the three axial directions in the same way as for the computations reported in Table 15.1 above; see Figure 15.6. By specifying sealing boundary conditions along the other boundaries, we ensure that the effective flow over the block follows the same axial direction as the pressure drop. This setup emulates how permeability is measured on core samples in the laboratory and provides us with three pairs of flow rates in 3D that can be used to compute an upscaled permeability tensor with diagonal elements

$$K_{xx} = -\frac{v_x L_x}{\Delta p_x}, \quad K_{yy} = -\frac{v_y L_y}{\Delta p_y}, \quad K_{zz} = -\frac{v_z L_z}{\Delta p_z},$$

where v_x is the net flux, L_x is the characteristic length of the block, and Δp_x is the pressure drop inside the block in the x -direction. With this method, the off-diagonal elements will always be zero by construction. Strictly speaking, the assumption of sealing boundaries is only valid in the idealized case when

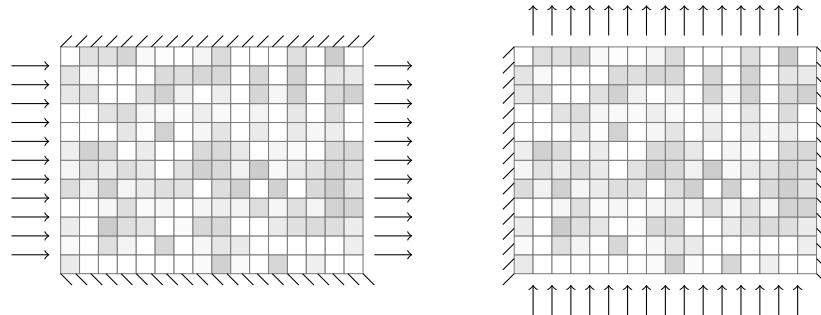


Fig. 15.6. Illustration of a simple flow-based upscaling method, solving $-\nabla \cdot (\mathbf{K} \nabla p) = 0$, with $p = 1$ and $p = 0$ prescribed along the inflow and outflow boundaries respectively, and no-flow boundary conditions elsewhere.

the permeability field is symmetric with respect to the faces of the coarse grid, i.e., if the coarse block is surrounded by mirror images of itself, and it can be shown that this method tends to introduce an upscaling bias towards low permeability values by thickening shale barriers and narrowing sand channels [132, 134].

Unless the grid block is located next to a sealing fault or an impermeable layer, it is therefore more natural to assume that the block has open boundaries so that a pressure differential applied along one of the axial directions will induce a net flow also in transverse directions. This means that the effective permeability will be a full tensor. One method to emulate open boundaries is to prescribe a constant pressure on faces perpendicular to each flow direction and linear pressure drop along the sides that are parallel to the flow direction [96, 224] so that the flow can leave or enter these sides. As for the sealing boundary, a unit pressure drop is applied in each of the axial directions. Use of linear boundary conditions is only strictly valid if the heterogeneous coarse block is embedded inside a homogeneous medium and tends to produce permeabilities with a bias towards high values [77].

Another popular option is to prescribe periodic boundary conditions [77], assuming that the grid block is sampled from a periodic medium so that the fluxes in and out of opposite boundaries will be equal. In other words, to compute the x -component of the permeability, we impose the following conditions

$$\begin{aligned} p(L_x, y, z) &= p(0, y, z) - \Delta p, & v(L_x, y, z) &= v(0, y, z), \\ p(x, L_y, z) &= p(x, 0, z), & v(x, L_y, z) &= v(x, 0, z), \\ p(x, y, L_z) &= p(x, y, 0), & v(x, y, L_z) &= v(x, y, 0). \end{aligned} \quad (15.12)$$

and similarly for the other axial directions. This approach gives a symmetric and positive definite tensor and is usually more robust than specifying sealing boundaries. Using periodic boundaries tends to give permeabilities that lie in between the lower and upper bounds computed using sealing and linear boundaries, respectively.

Let us see how two of these methods can be implemented in MRST for a rectilinear or corner-point grid. For simplicity, we assume that the d -dimensional domain to be upscaled is rectangular and represented by grid G and rock structure rock . We start by setting up structures representing boundary conditions,

```
bcsides = {'XMin', 'XMax'; 'YMin', 'YMax'; 'ZMin', 'ZMax'};
for j = 1:d;
    bcl{j} = pside([], G, bcsides{j, 1}, 0);
    bcr{j} = pside([], G, bcsides{j, 2}, 0);
end
Dp = {4*barsa, 0};
L = max(G.faces.centroids)-min(G.faces.centroids);
```

The first structure, `bcsides`, just contains name tags to locate the correct pair of boundary faces for each flow problem. The `bcl` and `bcr` structures are used to store a template of all the boundary conditions that will be used. While this is not needed in order to implement the pressure-drop method, they will come in handy when setting up the periodic boundary conditions. Finally, `Dp` and `L` contain the prescribed pressure drop and the characteristic length in each axial direction. With the data structures in place, the loop that does the upscaling for the pressure-drop method is quite simple

```

for i=1:d
    bc = addBC( [], bcl{i}.face, 'pressure', Dp{1});
    bc = addBC(bc, bcr{i}.face, 'pressure', Dp{2});
    xr = incompTPFA(initResSol(G, 100*barsa, 1), G, hT, fluid, 'bc', bc);
    v(i) = sum(xr.flux(bcr{i}.face)) / sum(G.faces.areas(bcr{i}.face));
    dp(i) = Dp{1}/L(i);
end
K = convertTo(v./dp, milli*darcy);

```

That is, we loop over the axial directions and for each direction we: (i) specify a pressure from left to right, (ii) compute pressures and fluxes by solving the resulting Poisson problem, and (iii) compute average velocity `v` across the outflow boundary and the average pressure drop `d`. Finally, we can compute the effective permeability by inverting Darcy's law. The implementation described above is a key algorithmic component in steady-state upscaling of relative permeabilities for multiphase flow and is therefore offered as a separate utility function `upscalePermeabilityFixed` in the `upscale` module.

The case with periodic boundary conditions is slightly more involved. We start by calling a routine that modifies the grid structure so that it represents a periodic domain.

```

[Gp, bcp] = makePeriodicGridMulti3d(G, bcl, bcr, Dp);
for j=1:d, ofaces{j} = bcp.face(bcp.tags==j); end

```

Technically, the grid is extended with a set of additional faces that connect cells on opposite boundaries of the domain. The routine also sets up an appropriate structure for representing periodic boundary conditions, which we use to extract the faces across which we will later compute outflow. You can find details of how the periodic grid is constructed in the source code. With the modified grid in place, the loop for computing local flow solutions reads

```

dp = Dp{1}*eye(d); nbcp = bcp;
for i=1:d
    for j=1:d, nbcp.value(bcp.tags==j)=dp(j,i); end
    xr = incompTPFA(initResSol(Gp, 100*barsa, 1), Gp, hT, fluid, 'bcp', nbcp);
    for j=1:d
        v(j,i) = sum(xr.flux(ofaces{j})) / sum(Gp.faces.areas(ofaces{j}));
    end
end

```

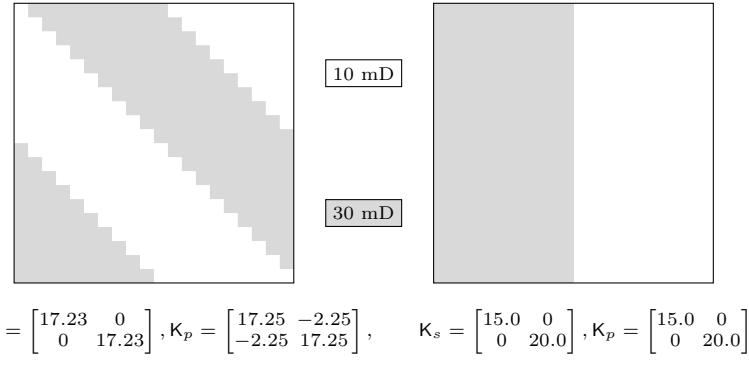


Fig. 15.7. Upscaling two isotropic permeability fields using flow-based upscaling with sealing boundaries (s) or periodic boundary conditions (p).

Inside the loop over all axial directions, the first for-loop extracts the correct pressure drop to be included in the periodic boundary conditions from the diagonal matrix `dp`; that is, a pressure drop along the current axial direction and zero pressure drop in the other directions. Then, we solve the flow problem and compute the average velocity across the outflow boundaries. Notice that because of the periodic conditions, we may have outflow also in the boundaries that have no associated pressure drop, which is why we may get a full-tensor permeability. Outside the loop, we compute the average pressure drop in each axial direction and invert Darcy's law to compute the permeability tensor

```
dp = bsxfun(@rdivide, dp, L);
K = convertTo(v/dp, milli*darcy)
```

The implementation described above is offered as a utility function called `upscalePermeabilityPeriodic` in the `upscale` module. Linear boundaries are not supported in MRST, but can be implemented by combining elements from the two cases discussed above: using the function `pside` to set a pressure distribution on the boundaries parallel to the flow direction and solving a small linear system to compute the components of the effective permeability.

Whereas setting up the necessary flow problems is relatively simple when the grid corresponds to just one coarse block, it is a bit more involved to do this upscaling efficiently for many blocks at a time. In the `upscale`, we therefore offer a utility function `upscalePerm` that computes permeability upscaling using the pressure-drop method.

Let us test the methods on a different permeability fields. Figure 15.7 shows two examples of upscaling using flow-based upscaling with sealing boundaries and periodic boundary conditions. For the case to the right, the correlation in the permeability field is along the y -direction, and both methods compute the same diagonal upscaled tensor. If we rotate the grid so that the correlation direction is along the diagonal, using pressure drop along the axial directions

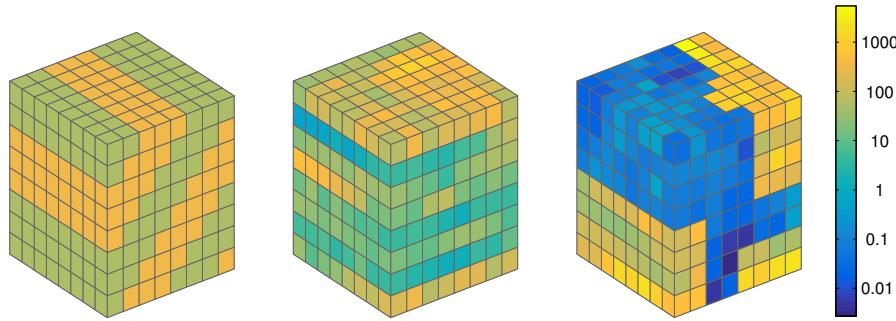


Fig. 15.8. Three models used to test the accuracy of averaging techniques. From left to right: an isotropic model with dipping layers and two subsets from the Tarbert and the Upper Ness formation, respectively, from the SPE 10 data set.

Table 15.2. Ratio between flow rate predicted by upscaled and fine-scale models for three different permeability fields, three flow scenarios, and three upscaling methods.

Model	Flow pattern	Harm-arith	Sealing	Periodic
Layered	East→West	0.78407	1.00000	1.02594
	North→South	0.49974	1.00000	1.00000
	Corner→Corner	1.04273	1.30565	1.34228
Tarbert	East→West	0.86756	1.00000	0.56111
	North→South	0.89298	1.00000	0.53880
	Corner→Corner	0.00003	0.00027	39.11923
Upper Ness	East→West	0.83026	1.00000	0.40197
	North→South	0.71283	1.00000	0.28081
	Corner→Corner	0.09546	0.62377	2183.26643

with sealing boundaries still gives a diagonal tensor, whereas the periodic boundary conditions gives a full tensor. By computing the eigenvalue decomposition of this tensor, we find that the upscaled tensor is diagonal with values 15 and 19.5 if we rotate the axial directions 45 degrees clockwise. Full source code for this example can be found in the `permeabilityExample1` script.

Let us also repeat the experiment from Table 15.1 on page 561. However, since both flow-based methods will correctly reproduce flow along or orthogonal to layered media by design, we replace the layered permeability field with one having a dipping layering. Moreover, we replace the top-to-bottom pressure drop by a pressure drop between diagonally opposite corners as well as changing the sampling points from the SPE 10 model. Figure 15.8 shows the new permeability fields and Table 15.2 reports the results of the experiment. Whereas the method with sealing boundary conditions is exact by design for the first two flow fields, it gives incorrect flow rates for the diagonal flow, in particular for the Tarbert subsample. With periodic boundary conditions, we

get the exact flux when the flow direction follows regions of constant permeability from the north to the south side for the isotropic case. For the other two flow patterns, the flow directions cross dipping layers and we thus get minor deviations in the lateral direction and somewhat larger deviation for the diagonal flow. For the anisotropic subsamples, the periodic conditions generally give less accurate results than the other two methods, except for the diagonal flow on the Tarbert case.

COMPUTER EXERCISES:

102. The models in the `BedModels1` and `BedModel2` are examples of fine-scale rock models that are developed to for use in a workflow that propagates petrophysical properties from the core scale to the reservoir scale. Use the methods presented above to upscale the absolute permeability in these models.
103. Implement a method for upscaling with linear boundary conditions.
104. Estimates computed by the averaging and flow based methods have a certain ordering, from low to high values: harmonic, harmonic-arithmetic, sealing boundaries, periodic boundaries, linear boundaries, arithmetic-harmonic, and arithmetic. Make a small test suite of heterogeneity permeability fields and verify this claim.

15.4 Upscaling transmissibility

In the discrete case, the choice of an appropriate upscaling method depends on the numerical stencil used for the spatial discretization. In previous chapters we have seen that the two-point finite-volume method is the method of choice in reservoir simulation. When using this method, grid-block permeabilities are only needed to compute transmissibilities between neighboring coarse blocks. It would therefore be more convenient if we instead of computing an effective permeability tensor associated with each coarse block could compute the coarse transmissibilities associated with the interface between pairs of neighboring coarse blocks directly. These transmissibilities should be defined so that they reproduce fine-scale flow fields in an averaged sense. That is, instead of upscaled block-homogenized tensors \mathbf{K}^* , we seek block transmissibilities T_{ij}^* satisfying

$$v_{ij} = T_{ij}^* \left(\frac{1}{|\Omega_i|} \int_{\Omega_i} p d\vec{x} - \frac{1}{|\Omega_j|} \int_{\Omega_j} p d\vec{x} \right), \quad (15.13)$$

where $v_{ij} = - \int_{\Gamma_{ij}} (\mathbf{K} \nabla p) \cdot \vec{n} d\nu$ is the total Darcy flux across Γ_{ij} .

The upscaled transmissibilities T_{ij}^* can be computed in much the same way as we used to compute upscaled permeabilities above, see Figure 15.9. Here, we use a pressure drop to drive a flow across the interface Γ_{ij} between

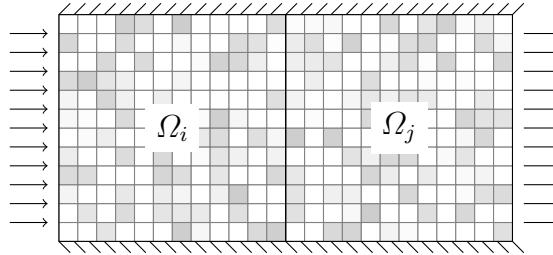


Fig. 15.9. Illustration of flow-based upscaling of transmissibility.

two coarse blocks Ω_i and Ω_j . Thus, by solving (15.5) in the two-block domain $\Omega_i \cup \Omega_j$ subject to suitable boundary conditions, we can compute the average pressures P_i and P_j in Ω_i and Ω_j and then obtain T_{ij}^* directly from the formula

$$v_{ij} = T_{ij}^*(P_i - P_j). \quad (15.14)$$

Let us see how this upscaling can be implemented in MRST. That is, we discuss how to upscale the transmissibility associated with a single interface between two grid blocks. To this end, let us assume that we have a fine grid G and a coarse partition vector q that subdivides the grid into two coarse blocks in the x -direction. We start by constructing a coarse-grid structure, as introduced in Section 14.2,

```
CG = generateCoarseGrid(G, q);
CG = coarsenGeometry(CG);
```

Using the data members in this structure, we can find all faces in the fine grid that lie on the interface between the coarse blocks as well as determining the correct sign to apply to the coarse-scale flux

```
i = find(~any(CG.faces.neighbors==0,2));
faces = CG.faces.fconn(CG.faces.connPos(i):CG.faces.connPos(i+1)-1);
sgn = 2*(CG.faces.neighbors(i, 1) == 1) - 1;
```

As we have seen above, there are different ways we can set up a localized flow problem that will give a flux across the interface between the two blocks. Assuming that the coarse interface will be more or less orthogonal to the x -axis, we use a pressure drop in this direction and no-flow boundary conditions on the other sides as shown in Figure 15.9:

```
bc = pside([], G, 'XMin', Dp(1));
bc = pside(bc, G, 'XMax', Dp(2));
xr = incompTPFA(initResSol(G, 100*barsa, 1), G, hT, fluid, 'bc', bc);
flux = sgn * sum(xr.flux(faces));
mu = fluid.properties();
```

All that now remains is to compute pressure values P_i and P_j associated with each coarse block before we can use (15.14) to compute the effective

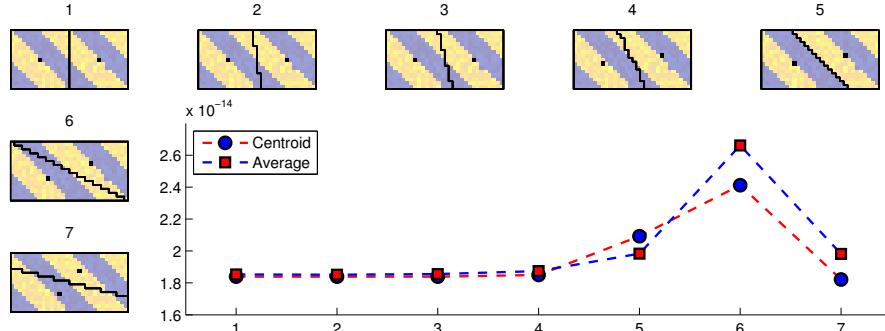


Fig. 15.10. Upscaled transmissibility computed on a series of two-block configurations using either centroid values or averaged values for the block pressures. Black dots show the location of the block centroids from which pressure values are sampled.

transmissibility Following (15.13), P_i and P_j should be defined as the average pressure inside each block

```
P = accumarray(q,xr.pressure) ./accumarray(q,1);
T = mu*flux/(P(1) - P(2));
```

Here, we have assumed that all grid cells have the same size. If not, we need to weight the pressure average by cell volumes. As an alternative, we could also have used the pressure value at the block centroids, which generally will give a (slightly) different transmissibility

```
cells = findEnclosingCell(G,CG.cells.centroids);
P      = xr.pressure(cells);
```

In Figure 15.10 we have used the code above to compute the transmissibility between two coarse blocks that cover a rectangular domain $[0, 200] \times [0, 100]$ m². We gradually rotate the interface between the two coarse blocks so that it goes from vertical towards being horizontal. Complete source code for the example is found in the script `transmissibilityExample1` in the `book` module. For the first four pairs of coarse blocks shown in Figure 15.10, there is only a slight difference between the transmissibilities computed using pressure values defined as block averages and pressures defined by sampling at the block centroids. The transmissibilities start to deviate as the block interface approaches the diagonal. On the other hand, whereas a pressure drop along the x -axis is a reasonable drive mechanism for the first block pairs, it is highly questionable for the last two block pairs. Let us, for instance, consider block pair number seven. If we instead apply the pressure drop along the y -axis to create a flow that is more perpendicular to the coarse interface, the transmissibility computed with block-averaged pressures changes from 2.0e-14 to 8.1e-14, whereas the value is 8.9e-14 with a pressure drop along both axial directions. (The latter setup is generally not well-posed as it will have singularities at the

south-east and north-west corners). Depending upon what type of flow conditions the block pair will be subject to in subsequent simulations, it might be better to set a pressure drop orthogonal to the coarse-block interface or use one of the oversampling methods that will be introduced in the next section.

An important property of the upscaled transmissibilities T_{ij}^* is that they all should be positive across all interfaces that can transmit fluids since this will ensure that the TPFA scheme defined by

$$\sum_j T_{ij}^*(p_i - p_j) = q_i,$$

will reproduce the net grid block pressures $p_\ell = \frac{1}{|\Omega_\ell|} \int_{\Omega_\ell} p d\vec{x}$, and hence also the coarse fluxes v_{ij} . Unfortunately, there is no guarantee that the transmissibilities defined by (15.13) are positive, e.g., if the blocks are such that the chord between the cell centroids does not cross the coarse interface. However, negative transmissibilities may also occur for regular block shapes if the permeability is sufficiently heterogeneous (e.g., as in the SPE 10 model). Even worse, unique transmissibility values may not even exist, since the upscaling problem generally is not well-posed, see [262] for a more thorough discussion of existence and uniqueness. To guarantee that the resulting coarse-scale discretization is stable, one should ensure that the T_{ij}^* values are positive. A typical trick-of-the-trade is to set $T_{ij} = \max(T_{ij}, 0)$, or in other words, just ignore ill-formed connection between neighboring grid blocks. This is generally not a satisfactory solution, and one should therefore either try to change the grid, use different pressure points to define the transmissibility, or apply some kind of fall-back strategy that changes the upscaling method locally.

The `upscaling` module does not contain any simple implementation of transmissibility upscaling like the one outlined above. Instead, the module offers a routine, `upscaleTrans`, that is designed for the general case where both the coarse and the fine grids can be fully unstructured and have faces that do not align with the axial direction. To provide robust upscaling for a wide range of geological models, the routine relies on a more comprehensive approach that will be outlined briefly in the next section and also has fallback strategies to reduce the number of negative permeabilities.

15.5 Global and local-global upscaling

The methods described so far in this chapter have all be local in nature. Averaging methods derive upscaled quantities solely from the local heterogeneous structures, whereas flow-based methods try to account for flow responses by solving local flow problems with prescribed boundary conditions. These boundary conditions are the main factor that limits the accuracy of flow-based methods. In Table 15.2 on page 566, we saw how the neither of the local methods were able to accurately capture the correct net flow for the

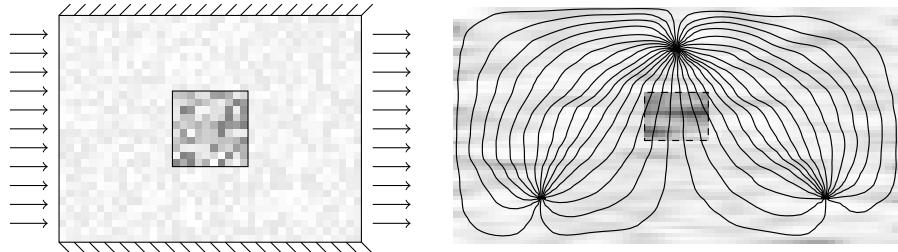


Fig. 15.11. Illustration of oversampling techniques for upscaling. The size of the grid block is exaggerated in the right plot.

anisotropic Tarbert and Upper Ness samples with boundary conditions giving diagonal flow. The main problem for flow-based techniques is, of course, that we do not know *a priori* the precise flow that will occur in a given region of the reservoir during a subsequent simulation. Thus, it is generally not possible to specify the appropriate boundary conditions for the local flow problems in a unique manner unless we already have solved the flow problem.

Improved accuracy can be obtained if you use a so-called *oversampling technique* which is sometimes referred to as an overlapping method. In oversampling methods, the domain of the local flow problem is enlarged with a border region that surrounds each grid block and boundary conditions, or other mechanisms for driving flow like wells or source terms, are specified in the region outside the domain you wish to upscale. The flow problem is then solved in the whole enlarged region, but the effective permeability tensor is only computed inside the original coarse block. This way, you lessen the impact of the localization assumption and the particular mechanism used to drive flow, and the flow across the boundaries of the coarse block is to a larger extent determined by the local heterogeneity surrounding the block. The motivation for using an oversampling technique is to better account for permeability trends that are not aligned with the grid directions and possible large-scale connectivity in the permeability fields.

To also account for global flow patterns, one can solve the pressure equation one time, or a few times, on the full geological model and sample fluxes from the global flow field and impose them as boundary conditions on the local flow problem used to upscale permeabilities or transmissibilities. In these so-called *global upscaling methods*, the pressure solution can either be computed using representative and *specific* drive mechanisms or the pressure solution can be computed using a set of *generic conditions* that e.g., flow the reservoir from east to west, north to south, and so on. From a computational point of view, this approach may seem to contradict the purpose of upscaling, but can be justified for numerical simulation of compressible and/or multiphase flow. Indeed, for such transient flows the pressure equation has to be solved multiple times throughout the simulation, and the cost of solving the global

pressure equation once (or even a few times) on a fine grid will in many cases be small compared with the cost of

Computing a global pressure solution can be avoided if one uses a so-called local-global method [55, 54, 93]. In these methods, pressure solutions obtained by solving global flow problems on a coarse grid are used to set boundary conditions for local flow problems that determine the upscaled transmissibilities. Since the initial coarse-grid computation may give quite poor boundary conditions for the local flow problems, these methods use a iterative bootstrapping procedure to gradually ensure consistency between the local and the global calculations.

Global and local-global upscaling methods may suffer from negative or anomalously large transmissibility values. To avoid these, Holden and Nielsen [109], who were among the first to study global upscaling methods, proposed an iterative process involving the solution of an optimization problem, that gradually perturbs the transmissibilities until they fall within a prescribed interval. A similar technique that avoids the solution the fine-grid problem was presented in [192]. Chen and Durlofsky [54] observed that unphysical transmissibilities occur mainly in low-flow regions. Therefore, instead of using an optimization procedure that alters all transmissibilities in the reservoir, they proposed to use a thresholding procedure in which negative and very large transmissibilities are replaced by transmissibilities computed by a local method. Because the transmissibilities are altered only in low-flow regions, the perturbation will have limited impact on the total flow through the reservoir.

The script `upscaleTrans` offers various implementations of global transmissibility upscaling using either specific or generic mechanisms to drive flow in the full reservoir model. Whereas the generic approach enables you to tailor-make the coarse-scale model for a particular simulation setup, the generic approach will give you a more robust upscaling that is less accurate for a specific simulation setup, but need not be recomputed if you later wish to simulate setups with significant changes in the well pattern, aquifer support, and other factors that affect the global flow patterns in the reservoir. Local-global methods, however, are not yet supported in MRST. Instead, the software offers various multiscale methods [81]—including multiscale mixed finite elements [5, 207], multiscale finite volumes [175, 177], and multiscale restriction-smoothed basis [178, 179]—offering an alternative approach in which the impact of fine-scale heterogeneity is included more directly into coarse-scale flow equations.

COMPUTER EXERCISES:

105. Extend the computations reported in Table 15.2 to include oversampling methods. Does this improve the accuracy of the two flow-based methods?
106. Implement a function that performs generic global upscaling for uniform partitions of rectilinear and curvilinear grids. To drive the global flow pat-

tern, you can use a pressure drop in each of the axial directions. How would you extend your method to more general grids and partitions?

107. Extend the `upscaling` module to include local-global upscaling methods.

15.6 Upscaling examples

In this section we will go through two examples where the methods discussed above are used to upscale full reservoir models. In doing so, we will also introduce you to a simple form of flow diagnostics [230, 184] that can be used to assess the accuracy of a single-phase upscaling.

15.6.1 Flow diagnostics quality measure

As we have mentioned several times already, a challenging problem with upscaling is to predict upfront whether the upscaling will be accurate or not. To partially answer this question and give an indication of the quality of the upscaling, we suggest to compare the cumulative well-allocation factors computed using the fine-scale and the upscaled models. As shown in Section 13.1.3 on page 502, a well-allocation factor is the percentage of the flux in or out of a completion that can be attributed to a pair of injection and production wells.

To compute these factors, we need to first compute a global, single-phase pressure solution for the specific well pattern we want to investigate. We then use the resulting flow field to compute numerical partitions (influence regions) as discussed in Sections 4.3.4 and 5.3 that can subsequently be used to subdivide the reservoir volume into subvolumes associated with pairs of injection and production wells as discussed in Section 13.1. Global methods tend to compute at least one fine-scale pressure solution as part of the upscaling procedure, but if a global solution is not available for the specific well pattern, it is generally not very expensive to compute compared with the cost of the flow-based upscaling procedure. The same goes for the computation of a coarse-scale flow field and partition functions on the fine and the upscaled model.

To improve the predictive power of well-allocation factors, we may have to subdivide each well into two or more segments that each consist of a connected set of completion cells. This way, we can measure the communication between different parts of the wells, which is particularly important if the model to be upscaled includes layers or geological objects with significantly different permeabilities.

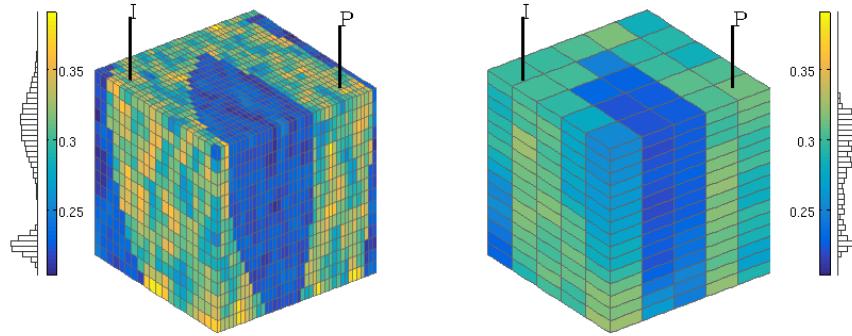


Fig. 15.12. Porosity distribution in a model with two different rock types before and after upscaling.

15.6.2 Model with two rock types

We start by considering a small rectangular model that contains two facies with contrasting petrophysical properties. An injector and a producer are placed diagonally opposite each other and completed mainly in the high-permeable parts of the model. Both wells operate at a fixed rate that amounts to the injection/production of 0.2 pore volumes per year, and to avoid potential problems with cross-flow, gravity forces are neglected. The fine grid consists of $40 \times 20 \times 15$ cells, which we seek to upscale to a coarse $5 \times 5 \times 15$ model. That is, we upscale by a factor 8×4 in the lateral direction and leave the vertical layers in an attempt to preserve the vertical communication in the model as accurately as possible. To this end, we use the `upscalePerm` function, which upscales permeability using the flow-based method from Section 15.3.2 with sealing boundary conditions. Figure 15.12 shows the porosities of the fine and the coarse model. Complete source code for this example can be found in the script `upscaleExample1` of the book module.

To get an indication of the accuracy of this upscaling, we will look at the cumulative well-allocation factors introduced above. For this particular model, we have only one producer and one injector, and to better measure the communication within the reservoir, we subdivide each well into two segments, an upper and a lower segments, so that we altogether will have four well-pairs whose well-allocation factors can be used to verify the quality of the upscaling. The 3D plots in Figure 15.13 show the injection and volumetric partitions defined as regions with partition values larger than 0.5. The resulting well-allocation factors are shown in Figure 15.14. For each well segment, we plot a bar chart of the cumulative flux in/out of the completions that make up the segment, from bottom to top. In the plots, each well segment is assigned the same unique color that was used in Figure 15.13 and each bar is subdivided

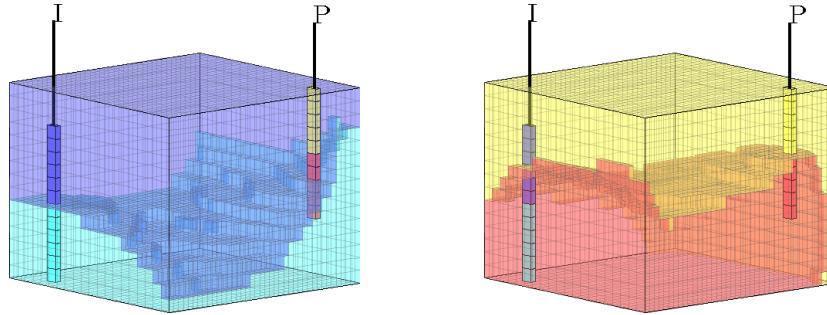


Fig. 15.13. Volumetric subdivision into injector-producer pairs by majority vote. The left plot shows the regions flooded by the upper and lower parts of the injector shown in blue and cyan, respectively. The right plot shows the drainage regions of the upper and lower half of the producer shown in yellow and red, respectively. In both plots, each cell is assigned to the completion that has the highest concentration value.

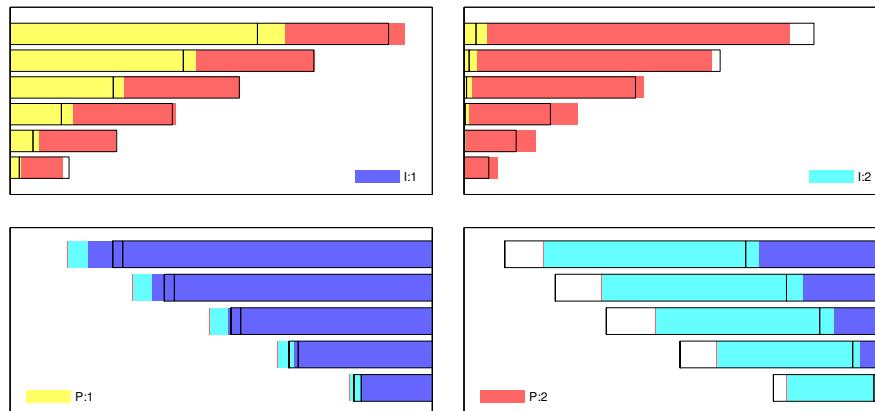


Fig. 15.14. Normalized cumulative well-allocation factors for the four well completions in the two-facies model. The color bars show well-allocation factors computed on a coarse model upscaled using `upscalePerm`, while the black lines show the same factors computed on the original model. The colored bars and the black lines should coincide if the upscaled model reproduces the flux allocation correctly.

into the fraction of the total in/outflux that belongs to the different well-pairs the segment is part of.

The bar charts at the top of Figure 15.14 show the allocation factors computed by the upscaled model for the upper half of the injector (I:1, upper-left plot), the lower part of the injector (I:2, upper-right plot), the upper part of the producer (P:1, lower-left plot), and the lower part of the producer (P:2, lower-right plot). The solid lines denote the same quantities computed for the

fine-scale model. The closer the color bars and the solid lines are, the better is the upscaling.

To explain how this type of flow diagnostics should be interpreted, let us look at the lower-left bar chart, which reports the allocation factors or the upper-half of the injector ($P:1$, yellow color). Here, the blue color signifies the flux that can be attributed to $I:1$, i.e., the upper-half of the injector, while the cyan color signifies the flux that can be attributed to $I:2$. Because the colored bars are mostly blue, the inflow into the upper half of the injector is predominantly associated with the $I:1-P:1$ pair. In other words, $P:1$ is mainly supported by the upper half of the injector and hardly in contact with the lower part. This trend is confirmed by looking at the upper-right bar chart, which shows that the inflow in the lower half of the injector mainly contributes to production in the lower part of the injector. The flooded volumes shown to the left in Figure 15.13 confirm that fluid injected from $I:2$ will mainly sweep the volume surrounding the lower of $I:2$. Similarly, the drainage regions in the plot to the right show that the drainage region of $P:1$ only engulfs the upper parts of $I:1$. The flooded regions of $I:2$, on the other hand, engulfs both $P:2$ and the lower parts of $P:1$, and hence the segment of the well will contribute flux to both segments of the producer, as shown in the upper-left chart of Figure 15.14. Likewise, the drainage region from $P:2$ engulfs both $I:2$ and the lower part of $I:1$ and hence is supported by flux from both the injector segments, as shown in the lower-right chart of Figure 15.14.

So, what does the flow diagnostics tell us about our upscaled model? First of all, the upscaling does obviously not fully reproduce the volumetric connections of the fine-scale model. Looking at well-allocation factors for the producers in the bottom row of Figure 15.14, we see that the upscaled model predicts too much flux in the upper segment ($P:1$, yellow color, left chart) since the colorbars exceed the solid lines at the top. Likewise, in the lower segment of the producer ($P:2$, red color, right chart), we get too low flux. Altogether, the allocation factors seem to suggest that the main problems are that the upscaled model misses some of the vertical communication and exaggerates the connection between $I:1$ and $P:1$. We could obviously have used a single number to measure the overall discrepancy in flux allocation, but in our opinion you will benefit more from a graphical presentation like this as it may help you to identify the cause of the mismatch between the upscaled and the original model.

To develop a better coarse model, we can upscale transmissibilities and well indices using a global method. For a given flow scenario, like the one above with fixed well positions and constant well controls, one can determine a set of transmissibilities and well indices that will reproduce a single pressure step exactly. This may seem desirable, but has the disadvantage that the linear system becomes negative definite, which in turn may lead to unphysical solutions away from the scenario used for upscaling. To span a reasonable set of global flow directions, we will instead use generic boundary conditions that pressurize the reservoir from east to west, from north to south, and from top

to bottom. There are several different ways these three global flow fields can be combined to compute a transmissibility values for each coarse face, see e.g., [125]. Here, we use a simple strategy suggested by [140], in which we pick the one among the three flow field that gives the largest flux across a given coarse face and insert this into (15.14) to compute the transmissibility associated with this face. A rationale for this choice is that the accuracy of the coarse model is most sensitive to the choice of transmissibilities in high-flow regions.

The well indices are upscaled using the following formula (see (4.27) on page 128)

$$J_i^* = Q_i / (P_i - p_w) \quad (15.15)$$

where Q_i is the sum over rates of all the perforations inside block i , P_i is the average block pressure, and p_w is the pressure inside the well. To obtain representative flow fields, we solve $n_w - 1$ pressure equations, where n_w is the number of wells. Each solution is computed by setting a positive pressure in one well and zero pressure in all the other wells. To combine the flow fields, we can either use the same strategy as for the transmissibilities and pick the solution that gives the largest well rate, or as we will do in the current example, add all well rates and divide by the sum of the corresponding pressure drops.

The approach outlined above has been implemented in the function `upscaleTrans`. Unfortunately, there is no guarantee that the upscaled transmissibilities are not negative. To avoid creating an ill-conditioned discretization matrix, we therefore set all negative transmissibilities to zero, thereby blocking the corresponding coarse face for flow. This is in our experience an acceptable solution as long as the upscaled transmissibilities are only intended for a specific scenario, like herein, since negative transmissibilities tend to appear because flow is tangential to the coarse interface or as a result of numerical noise in regions with very low flow. If your upscaling, on the other hand, aims to serve a more general purpose with varying well patterns and/or boundary conditions, you would need to implement a more sophisticated strategy that falls back to a local flow-based method, an averaging method, or a combination of these.

Figure 15.15 shows the resulting match in well-allocation factors, which is significantly better than when using a simple permeability upscaling. In part, this is a result of improved transmissibilities, and in part because we now have upscaled the well indices.

15.6.3 SPE 10 with six wells

You have already encountered Model 2 from the 10th SPE Comparative Solution Project multiple times throughout the book. Because of its simple grid and strong heterogeneity and the fact that it is freely available online, this data set has become a community benchmark that is used for many different purposes. The original aim of the project was to “compare upgridding and upscaling approaches and the ability to predict performance of a waterflood

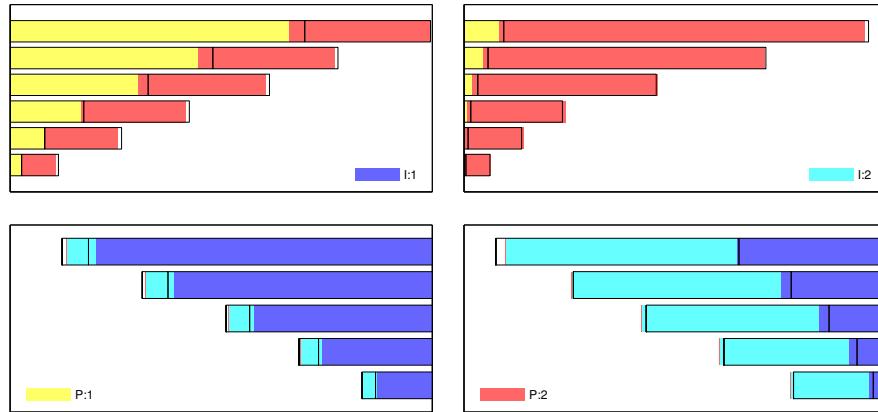


Fig. 15.15. Normalized cumulative well-allocation factors for the four well completions in the two-facies model. The color bars show well-allocation factors computed on a coarse model upscaled using `upscaleTrans`, while the black lines show the same factors computed on the original model.

through a million cell geological model”, and in [62] you can read about the relative merits of the various methods used in the upscaling studies that were submitted to the project by August 2000. In short, the study showed that the data set is generally very difficult to upscale accurately using single-phase methods and that the best results are generally obtained by methods that also account for multiphase effects. Later, however, quite good results have been obtained using local-global and multiscale methods, see e.g., [2].

In this example, we will compare the ability of four different upscaling methods to accurately predict single-phase flow. That is, we consider two averaging methods (harmonic and harmonic-arithmetic) and a local flow-based upscaling with sealing boundary conditions for upscaling permeability, and the global upscaling method discussed at the end of the previous example for upscaling transmissibilities and well indices. (However, unlike in the previous example, we use maximum flow rate to determine both transmissibilities and well indices). Full source code can be found in the script `upscaleExample2`.

We start by considering the Tarbert formation, which can be found in the upper 35 layers of model. To upscale this Cartesian grid model, we choose a coarsening factor of $10 \times 10 \times 3$, and to get an even number of cell layers inside each grid block, we extend the model slightly by repeating the top layer so that the fine-scale model altogether has $220 \times 60 \times 36 = 475\,200$ cells. To get a slightly more interesting well pattern, we replace the original five-spot pattern with a pattern consisting of producers in the four corners of the model and two injectors located near the center of the model, see Figure 15.16. All four wells operate at a fixed bottom-hole pressure of 200 bar for the producers and 500 bar for the injectors. In Figure 15.17, we verify whether the well-allocation

rates predict by the four upscaling methods for injector I2 reproduce the fine-scale results or not. This well has significant communication with all four producers and hence is more difficult to match than injector I1, which mainly communicates with P3 and P4.

Harmonic averaging not surprisingly gives well rates that are far from the fine-scale targets. As explained in Section 15.3.1, the harmonic mean has a bias against smaller values and tends to underestimate the effective permeability. Since all wells are controlled by bottom-hole pressure, too low permeability means too low well rates. If the wells were controlled by total rate, as in the previous example, the well rates predicted by the harmonic average would have been more correct, but the predicted pressure build-up around the injector and the pressure draw-down in the producers would both be too high.

Comparing the local, flow-based method to the harmonic-arithmetic averaging method, we see that the latter is significantly more accurate. By design, the flow-based method should give accurate prediction of flow along the axial directions of the grid. Here, however, we have flow that is strongly affected by heterogeneity and mostly goes in the lateral, diagonal direction and hence the harmonic-arithmetic method seems to be more suited, in particular if we consider the computational time of the two methods. In our current implementation, the flow-based method extracts a subgrid and assembles and inverts three local matrices for each coarse block. The computational overhead of this procedure is significant since each new local solve incurs full start-up cost. The result is that the function `upscalePerm` has orders of magnitude higher computational cost than the harmonic-arithmetic averaging, which is implemented using a few highly efficient calls to `accumarray`.

Using the global upscaling method, we get almost exact reproduction of the fine-scale well-allocation factors. The plots may be a bit deceiving because of the areas where the colorbars extend beyond the solid lines representing the fine-scale well-allocation. However, bear in mind that the bars and lines represent *cumulative* factors, and thus we should only look at the discrepancy at the top of each colorbar, and if you look closely, you will see that here the match is excellent. There are two reasons for this: On one hand, using global flow field will ensure that each upscaling problem has more representative boundary conditions used for localization and hence can better account for correlations in the heterogeneity that extend beyond the block. On the other hand, the global method includes upscaling of well indices that can have a significant impact on the well rates that determine the flux allocation. In passing, we also note that although the computational cost of the global method is significantly lower than that of the local method, primarily since we here have been able to utilize a highly efficient multigrid solver for the global flow problems and avoid solving a long sequence of small problems that each have a large start-up cost.

For completeness, we also include the results of a similar study for the whole SPE 10 model, see Figure 15.18. this includes the fluvial Upper Ness formation, which is notoriously difficult to upscale accurately and we also see

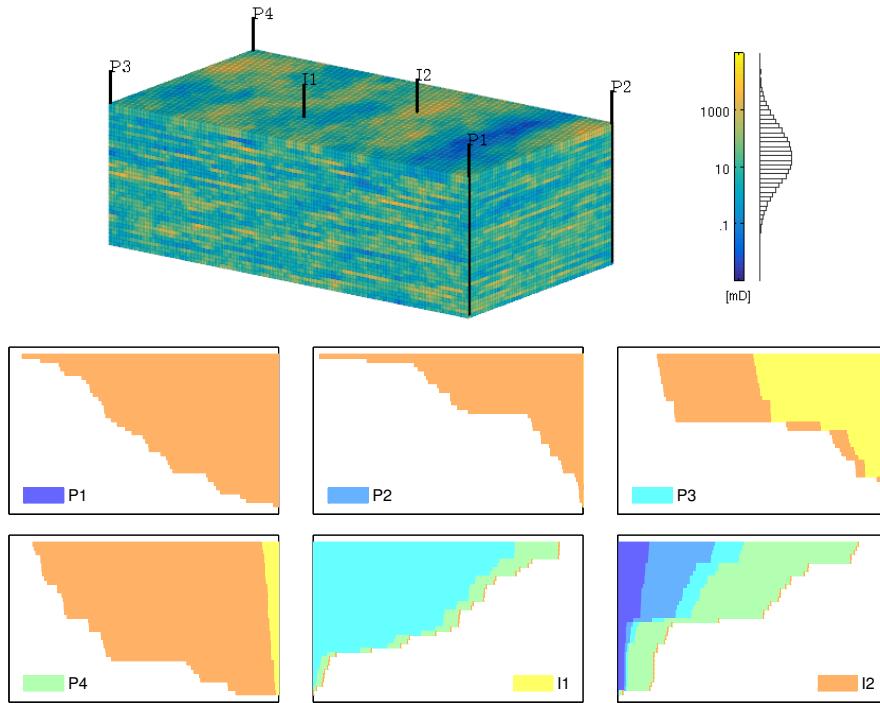


Fig. 15.16. Model of the Tarbert formation with six wells, two central injectors and four peripheral producers. The 3D plot shows lateral permeability distribution, while the bar charts shows normalized cumulative well-allocation factors.

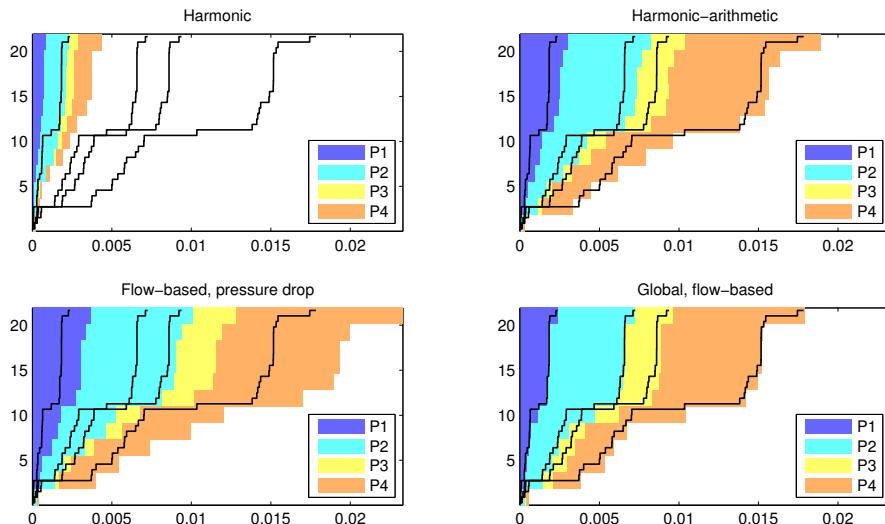


Fig. 15.17. Well-allocation factors for injector I2 in the Tarbert model computed on four different coarse models.

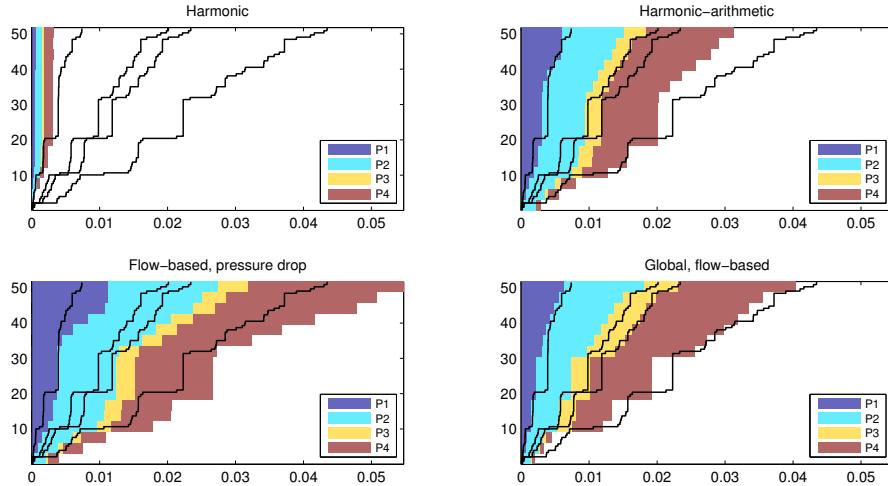


Fig. 15.18. Well-allocation factors for injector I2 in the full SPE 10 model computed on four different coarse models.

that the harmonic-arithmetic and the global method are no longer as accurate as for the Tarbert formation. For this 1.1 million grid cell model, an efficient iterative solver like AGMG is indispensable to be able to compute the fine-scale solution and perform a global upscaling in MATLAB.

15.6.4 General advice and simple guidelines

Through the various examples in this chapter, we have tried and hopefully succeeded to convince you that accurate and robust upscaling is a difficult problem and that there is no single upscaling method that is unequivocally better than the others. This is one motivation for the tremendous attention so-called multiscale methods have received in recent years since these methods offer a promise of a more systematic and consistent way of bringing the impact of small-scale heterogeneity variations into simulations on a coarser scale.

What is the best upscaling method for a specific project will depend on many factors. The most important factor is probably what you intend to use the upscaled model for. If you are upscaling for a specific scenario, it is strongly recommended to use a method that employs specific global information as this generally will enable you to perform more aggressive coarsening. If, on the other hand, the upscaled model is to be used to simulate a large variety of scenarios, or some yet unspecified scenario, you should pick a technique that is as robust as possible. What this technique is, will depend on the type of heterogeneity you are facing, the flow patterns that exist in a reservoir, and so on. For relatively homogeneous reservoirs with flow patterns that mainly follow the axial directions, a simple averaging technique may be sufficient, whereas techniques that utilize some kind of global information are

recommend for highly heterogeneous reservoirs with strong contrasts and large variations in correlation lengths. Other determining factors include the time and the computer resources you have available for upscaling, the number of petrophysical realizations you need to upscale, the required upscaling factor, etc.

Irregardless of your situation, it is generally recommended that you try different upscaling and grid coarsening methods. Start by simple averaging methods that have a low computational cost and can quickly give you a first estimate of the upscaling uncertainty or more generally the uncertainty in the petrophysical properties. Check to see if there are features in your model that need to be resolved by adapting the coarse grid and ensure that you do not create a coarse grid that will give problems for subsequent flow simulations, see e.g., the discussion in Section 14.4. Then you can gradually move towards more sophisticated upscaling methods. For local methods, you should try both sealing and linear pressure boundaries to provide lower and upper bounds. You may also need to check the use of oversampling methods to lessen the impact of boundary conditions used for localization.

At all stages, you should use representative single-phase simulations to validate your upscaled model against the original fine-scale model. To this end, we strongly recommend that you use various kinds of flow diagnostics like the well-allocation factors discussed in the examples above, partition functions that give you the volumetric connections between inflow and outflow, time-of-flight that gives time lines for displacement fronts, and so on. Herein, we have computed these quantities using finite-volume discretizations that are available in MRST, but these flow diagnostic measures can also be computed using streamline simulation [69], which is generally a very efficient tool for comparing geomodels and upscaled simulation models. Whether you choose one or the other, is not that important. What is important, is that you stick to using single-phase flow physics to avoid mixing in the effect of multiphase flow parameters like variations in relative permeabilities and capillary pressures, which need to be upscaled by other means.

A

The MATLAB Reservoir Simulation Toolbox

Practical computer modeling of porous media constitutes an important part of the book and is presented through a series of examples that are intermingled with more traditional textbook material. All examples discussed in the book rely on the MATLAB Reservoir Simulation Toolbox (MRST), which is a free, open-source software that can be used for any purpose under the GNU General Public License (GPLv3). The basic part of MRST contains a comprehensive set of data structures and routines for representing and manipulating the primary input parameters that make up a simulation model of a porous medium: grids representing geometry of the porous domain, petrophysical rock properties, and forcing terms such as gravity, boundary conditions, source terms, and well models. In addition, there are routines for reading and processing input files and plotting quantities defined over cells and cell interfaces, as well as functionality for automatic differentiation. On top of this, MRST provides a set of add-on modules that supply a wide range of discretizations, solvers, simulators, and workflow tools that can be combined to perform various tasks in reservoir modeling. By carefully documenting and releasing this software as free, open source, we hope to contribute to give a head start to students about to embark on a master or PhD project, as well as to researchers working on similar problems.

This chapter will provide you with a brief overview of MRST and the philosophy underlying its design. We show you how to obtain and install the software and explain its terms of use, as well as how we recommend that you use the software as a companion to the textbook. We also briefly discuss how you can use a scripted, numerical programming environment like MATLAB to increase the productivity of your experimental programming and give a few examples of tricks and ways of working with MATLAB that we have found particularly useful. We end the chapter by introducing you to automatic differentiation, which is one of the key aspects that make MRST a powerful tool for rapid prototyping and enable us to write compact and quite self-explanatory codes that are well suited for pedagogical purposes. As a complement to the material presented in this chapter, you should also

consult the first section [154] of the just-in-time online learning tools (Jolts) developed in collaboration with Stanford University. This Jolt gives a brief overview of the software, tells you why and how it was created, and shows you how to download and install it on your computer. If you are not interested in programming at all, you can jump directly to Chapter 2. However, if you choose to not work with the software alongside the textbook material, be warned, you might miss a lot of valuable insight.

A.1 Getting started with the software

The open-source MRST toolbox was originally developed to support 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 and verification on problems with a high degree of realism, and in many cases, full industry-standard complexity.

A.1.1 Core functionality and add-on modules

Because MRST is a research tool whose aim is to support research on modeling and simulation of flow in porous media, it contains a wide variety of mathematical models, computational methods, plotting tools, and utility routines. To make the software as flexible as possible, MRST is organized quite similar to MATLAB and consists of a collection of core routines and a set of add-on modules as illustrated in Figure A.1. The material presented in Part I of the book relies almost entirely on general routines from the core module, which contains routines and data structures for creating and manipulating grids, petrophysical data, and global drive mechanisms such as gravity, boundary conditions, source terms, and wells. The core also contains an implementation of automatic differentiation (you write the formulas and specify the independent variables, the software computes the corresponding derivatives or Jacobians) based on operator overloading (see Section A.5), as well as a few routines for plotting cell and face data defined over a grid. The functionality in the core module is considered to be stable and not expected to change significantly in future releases.

To minimize maintenance costs and increase flexibility, MRST core does not contain flow equations, discretizations, and solvers; these are implemented in various add-on modules. You have already encountered the `incompTPFA` solver from the `incomp` module in Section 1.4; this module implements standard solves for incompressible, immiscible, single-phase and two-phase flow. The mathematical models, discretizations, and solutions techniques underlying this module will be extensively discussed in Part II and III of the book. In

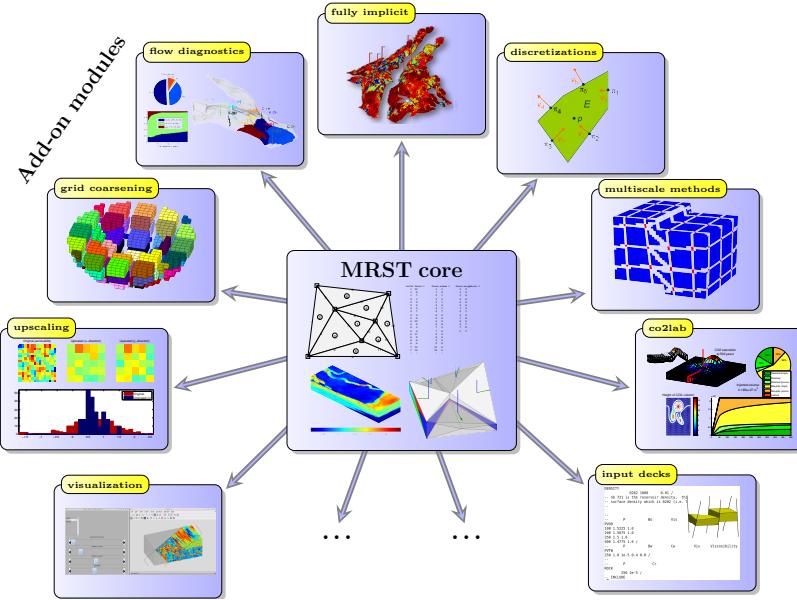


Fig. A.1. The Matlab Reservoir Simulation Toolbox consists of a core module that provides basic data structures and utility functions, and a set of add-on modules that offer discretizations and solvers, simulators for incompressible and compressible flow, and various workflow tools such as flow diagnostics, grid coarsening, upscaling, visualization of simulation output, and so on.

particular, Chapters 5 and 10 outline key functionality offered in the module and discuss in detail how the solvers are implemented. Chapter 6 introduces you to the `mimetic` and `mpfa` modules which offer consistent discretizations on general polyhedral grids [153]. Solvers for incompressible flow have been part of MRST since the beginning and constitute the first family of add-on modules signified by the 'Discretizations' block in Figure A.1. The modules `incomp`, `mimetic`, and `mpfa` are all implemented using an imperative programming model, i.e., using mathematical functions that operate mainly on vectors, (sparse) matrices, structures, and a few cell arrays. These are generally robust and well documented, have remained stable over many years, and will not likely change significantly in future releases. The family of incompressible flow solvers also include a few other discretization methods from the research front like virtual element methods and are intimately connected with two of the modules implementing multiscale methods (MsFV and MsMFE). These will not be discussed herein; details can be found in the software itself, in the MRST webpages, or in one of the many papers using the software.

The second family of modules consists of simulators based on automatic differentiation and is illustrated by the 'fully implicit' block in Figure A.1. The introduction of automatic differentiation in MRST has been a big success and

has both enabled efficient development of black-oil simulators, but also opened up for a unprecedented capabilities for rapid prototyping [139, 26]. The process of writing new simulators is hugely simplified by the fact that one no longer needs to compute analytic expressions for derivatives and Jacobians. We will come back to this in Chapters 7 and 11. Solvers using automatic differentiation were initially implemented using an imperative programming model similar to the `incomp` family of modules. Soon it became obvious that this was a limiting factor, and a new object-oriented programming model implementing a general framework for solvers referred to as MRST AD-OO [139, 25, 176, 26] was introduced. The individual modules and the underlying implementations of AD-OO have undergone significant changes in the period 2014–2016. From MRST 2016a and onward, however, we expect that the functionality will remain largely unchanged and only be subject to bugfixes, feature enhancements, and performance improvements.

The `ad-core` module is the most basic part in MRST AD-OO and does not contain any complete simulators, but rather implements the common framework used for many other modules. The design of this framework deviates significantly from that of the incompressible solvers. The main motivation for introducing an object-oriented framework is to be able to simulate compressible multiphase models of industry-standard complexity. In Chapter 11 at the end of Part III, we will discuss how to simulate compressible multiphase models of black-oil type, but future releases of MRST will also offer simulators for compositional flow. These types of multiphase models are significantly more complex to simulate than the basic incompressible models implemented in the `incomp` module. Not only do these models have more equations and more complex parameters and constitutive relationships, but making robust simulators also necessitates more sophisticated solution algorithms involving nonlinear solvers, preconditioners, time-step control, variable switching, etc. Moreover, industry-standard simulations generally require a number of bells and whistles to implement specific fluid behavior, well controls, etc., as well as a number of subtle tricks-of-the-trade to ensure that your simulator is able to reproduce results of leading commercial simulators. Using an object-oriented framework enables us to divide the implementation into different numerical contexts (mathematical model, nonlinear solver, time-step control, linearization, linear solver, etc), hide unnecessary details, and only expose the details that are necessary within each specific context.

The third family of modules consist of a series of tools that can be used as part of the reservoir modeling workflow. In Part IV of the book we will go through the three types of tools shown in Figure A.1. 'Diagnostics' signifies a family of computational methods for determining volumetric connections in the reservoir, computing well-allocation factors, measuring dynamic heterogeneity, providing simplified recovery estimates, etc. Tools from the 'grid coarsening' and 'upscale' modules can be used to develop reduced simulation models with fewer degrees of freedom and hence lower computational costs. MRST also offers several other modules of the same type, e.g., history

matching and production optimization, but these are outside the scope of the book.

The fourth family of modules consists of computational methods that have been developed to study a special problem. In Figure A.1, this is exemplified by the 'co2lab' module, which is a comprehensive collection of computational methods and modeling tools developed especially to study the injection and long-term migration of CO₂ in large aquifer systems. Other modules of the same type include solvers for geomechanics and various modeling frameworks and simulators for fractured media. These modules are all outside the scope of this book.

The last family consists of a variety of utility modules that offer graphical interfaces and advanced visualization, more comprehensive routines for reading and processing simulation models and other input data, C-acceleration of selected routines from the core module to avoid computational bottlenecks, etc. You will encounter functionality from several of these modules throughout the book, but the modules themselves will not be discussed in any detail.

A.1.2 Downloading and installing

The main parts of MRST are hosted as a collection of private software repositories on Bitbucket. Public releases are provided as self-contained archive files that can be downloaded from the webpage:

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

Assume now that you have downloaded the tarball of one of the recent releases; here, we use MRST 2016b as an example. Issuing the following command

```
untar mrst-2016b.tar.gz
```

in MATLAB will create a new folder `mrst-2016b` in your current working directory that contains all parts of the software. Once MRST has been extracted to some folder, which we henceforth refer to as the *MRST root* folder, you must navigate MATLAB there, either using the built-in file browser, or by using the `cd` command. Assuming that the files were extracted to the home folder, this would amount to the following on Linux/Mac OS,

<code>cd /home/username/mrst-2016b/</code>	% on Linux/Mac OS
<code>cd C:\Users\username\mrst-2016b\</code>	% on Windows

When you are in the folder that contains the software, you need to run the following command to activate it

```
startup;
```

The whole procedure of downloading and installing MRST, step by step, can be seen in the first MRST Jolt [154], which was developed using MRST 2014b.



Fig. A.2. The welcome message displayed when the startup script is run in MRST 2016a and later. The careful reader may notice that the user runs a development version of the software and not one of the official releases.

At this point, a word of caution is probably in order. Throughout the book, we will refer to the software as a toolbox. By this we mean that MRST is a collection of data structures and routines that can be used alongside with MATLAB. It is, however, *not* a toolbox in the same sense as those that can be purchased from the official vendors of MATLAB. This means, for instance, that MRST is not automatically loaded unless you start MATLAB from the MRST root folder, or make this folder your standing folder and manually issue the `startup` command. Alternatively, if do not want to navigate to the MRST root folder, for instance in an automated script, you can call startup directly

```
run /home/username/mrst-2016b/startup
```

In versions prior to MRST 2016a, the startup script only sets up the global search path so that MATLAB is able to locate MRST's core functionality and the various modules. To verify that the software is working, you can run the simple example discussed in the previous section by typing `flowSolverTutorial1`. This should produce the same plot as shown in Figure 1.3.

A.1.3 Exploring the functionality and getting help

In MRST 2016a and newer, the startup script will display a welcome message showing the software is initialized and ready to use, see Figure A.2. The welcome message contains links to a number of functions that are useful if you want to get more acquainted with MRST. Upon your first encounter with MRST, the best point to start is by typing (or clicking the corresponding blue text in the welcome message)

```
mrstExamples()
```

This will list all introductory examples found in MRST core. Some of these examples will introduce you to basic functionality in MRST, whereas others highlight capabilities found in various add-on modules that implement specific discretizations, solvers, and workflow tools. These examples are designed using cell-mode scripts, which can be seen as a type of “MATLAB workbook” that enables you to break a script down into smaller code sections (code cells) that can be run individually to perform a specific task such as creating parts of a model or making an illustrative plot; see Figure A.3 for an illustration. In our opinion, the best way to understand tutorial examples is to go through the corresponding scripts, evaluating one section at the time. Alternatively, you can set a breakpoint on the first line and step through the script in debug mode, e.g., as shown in the fourth video of the first MRST Jolt [154]. Some of the example scripts in MRST contain quite a lot of text and are designed to be published as HTML documents, as seen in to the right in Figure A.3. 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. If you want to know what a specific function (e.g., `computeTrans`) does, you can type

```
help computeTrans
```

which will bring up the documentation shown in Figure A.4. As a general rule, all core functionality is well documented in a format that follows the MATLAB standard. The same should apply to functionality you are meant to utilize from any of the add-on modules. (However, here the format and level of documentation may differ more, depending upon who developed the module.)

As a general rule, all modules distributed as part of the MRST release are required to contain worked tutorials highlighting key functionality that will be needed by most users; a subset of these tutorials is also available on the MRST webpage. To list tutorial examples found in individual modules, you can use the same command as above, which can also be used to list all the tutorial examples that the software offers,

```
mrstExamples('ad-blackoil')    % all examples in the black-oil module
mrstExamples('all')           % all examples across all available modules
```

To learn more about the different modules and get a full overview of all functionality that is available in MRST, you can type the command

```
mrstExploreModules()
```

This brings up a graphical user interface that lists all accessible modules and outlines their purpose and functionality, including a short description of all tutorial examples found in each module, as well as a list of relevant scientific publications, with possibility to view published and preprint versions and export citations to BibTeX. The modules will be discussed in more detail in Section A.3.

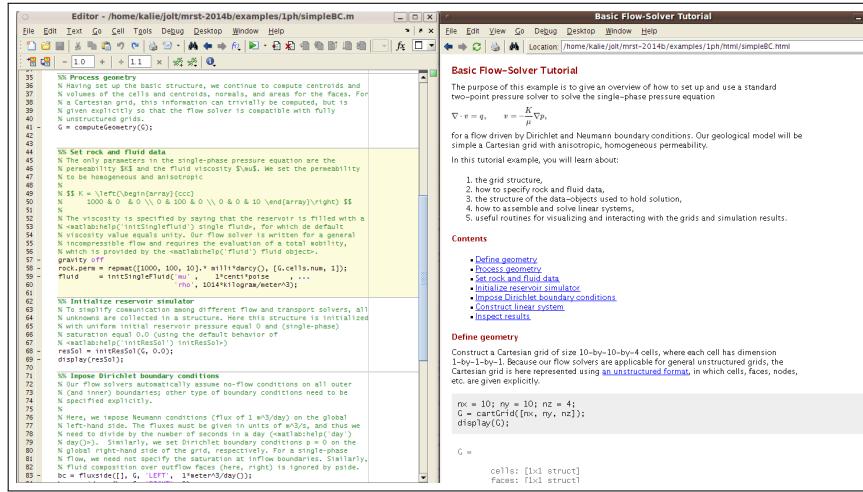


Fig. A.3. Illustration of the MATLAB workbook concept. The editor window shows part of the source code for the `simpleBC` tutorial from MRST 2014b. (In newer versions, this tutorial has been renamed `incompIntro` and moved to the new `incomp` module.) Notice how cells are separated by horizontal lines and how each cell has a header and a text that describes what the cell does. The exception is the first cell, which summarizes the content of the whole tutorial. The right window shows the result of publishing the workbook as a webpage.

```
>> help computeTrans
Compute transmissibilities.

SYNOPSIS:
T = computeTrans(G, rock)
T = computeTrans(G, rock, 'pn', pv, ...)

PARAMETERS:
G      - Grid structure as described by grid_structure.

rock - Rock data structure with valid field 'perm'. The permeability
       is assumed to be in measured in units of metres squared ( $m^2$ ).
       Use function 'darcy' to convert from darcies to  $m^2$ , e.g.,
           perm = convertFrom(perm, milli*darcy)
       if the permeability is provided in units of millidarcies.
:
:

RETURNS:
T - half-transmissibilities for each local face of each grid cell
   in the grid. The number of half-transmissibilities equals
   the number of rows in G.cells.faces.

COMMENTS:
PLEASE NOTE: Face normals are assumed to have length equal to
the corresponding face areas. ..

SEE ALSO:
computeGeometry, computeMimeticIP, darcy, permTensor.
```

Fig. A.4. Most functions in MRST are documented in a standard format that gives a one-line summary of what the function does, specifies the synopsis (e.g., how the function should be called), explains the input and out parameters, and points to related functions.

MRST also offers simplified access to a number of public data sets. To view a list of all these, you can use the following graphical user interface

```
mrstDatasetGUI()
```

which will briefly describe each data set, provide functionality for downloading and installing the data set in a standard location, list the files contained as well as the tutorial examples that use each data set. More details are given in Section A.2.

Last, but not least, a number of common queries have been listed on MRST's FAQ page:

<http://www.sintef.no/projectweb/mrst/faq/>

For further assistance and discussions about MRST you may visit (and subscribe to) the mailing list at

<http://www.sintef.no/projectweb/mrst/forum/>

or: sintef-mrst@googlegroups.com

A.1.4 Release policy and version numbers

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, one in the spring and one in the fall. 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. Likewise, parts of the software may sometimes be reorganized like when the basic incompressible solvers were taken out of MRST core and put in a separate module in 2015. In writing this, we (regretfully) acknowledge the fact that specific code details and examples in books describing evolving software tend to become somewhat outdated. To countermand this, complete codes for almost all examples presented in the book are contained in a separate `book` module that accompanies MRST 2015a and later releases.

The version number of MRST refers to the biannual release schedule and does not imply a direct compatibility with the same release number for MATLAB. That is, you do not need to use MRST 2013a if you are using MATLAB R2013a, or vice versa, you do not need to upgrade your MATLAB to R2016b to use MRST 2016b.

A.1.5 Software requirements and backward compatibility

MRST was originally implemented so that the minimal requirement should be MATLAB version 7.4 (R2007a). However, certain parts of the software use features that were not present in R2007a:

- The functionality for automatic differentiation uses new-style classes (`classdef`) that were introduced in R2008a.
- Various scripts and examples use new syntax for random numbers introduced in R2007b.
- Some script may use the tilde operator to ignore return values (e.g., `[~, i]=max(X, 1)`) that was introduced in R2009b.
- Some routines, like the fully implicit simulators for black-oil models, rely on accessing sub-blocks of large sparse matrices. Although these routines will run on any version from R2007a and onward, they may not be efficient on versions older than R2011b.

When it comes to visualization, things are a bit more complicated since MATLAB 3D graphics does not behave exactly the same on all platforms. Moreover, MATLAB introduced new handle graphics in R2014b, which has been criticized by many because it is slow and because it breaks backward compatibility. We have tried to revise plotting in newer versions of MRST so that it works well both with the old and the new handle graphics, but every now and then you may stumble across certain tricks (e.g., setting grid lines to be semi-transparent to make them thinner) may not work well on your particular MATLAB version.

Large parts of MRST can also be used with GNU Octave, which is an open-source numerical programming environment that is designed to be compatible with MATLAB. However, there are two main difficulties: GNU Octave has less (stable) functionality for 3D visualization, which is used a lot throughout this book, and does not yet offer the new-style classes (`classdef`) used in the implementation of automatic differentiation.

Although MRST is designed to only use functionality available in standard MATLAB, there are a few third-party packages and libraries that we have found to be quite useful:

- **MATLAB-BGL:** MATLAB does not yet have extensive support for graph algorithms. The Boost Graph Library (BGL) is a generic interface for traversing graphs. The MATLAB Boost Graph Library contains binaries for useful algorithms in BGL such as depth-first search, computation of connected components, etc. MATLAB-BGL is freely available under the BSD License from the Mathwork File Exchange¹. MRST has a particular module, see Section A.3, that downloads and installs this library.
- **METIS:** is a widely used library for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices [127]. The library is released² under a permissive Apache License 2.0.

¹ <http://www.mathworks.com/matlabcentral/fileexchange/10922>

² <http://glaros.dtc.umn.edu/gkhome/metis/metis/overview>

- **AGMG:** For large problems, the linear solvers available in MATLAB are not always sufficient, and it may be necessary to use an iterative algebraic multigrid method. AGMG [203] has MATLAB bindings included and was originally published as free open-source. The latest releases have unfortunately only offered free licenses for academic research and teaching³.

MATLAB-BGL is required by several of the more advanced solvers that are not part of the basic functionality in MRST. Installing the other two packages is recommended but not required. When installing extra libraries or third-party toolboxes that you want to integrate with MRST, you must make the software aware of them. To this end, you should add a new script called `startup_user.m` and use the built-in command `mrstPath` to make sure that the routines you want to use are on the search path used by MRST and MATLAB to find functions and scripts.

A.1.6 Terms of usage

The MRST software is distributed as free, open-source software under the GNU Public License (GPLv3)⁴. This license is a widely used example of a so-called copyleft license that offers the right to distribute copies and modified versions of copyrighted creative work, provided the same rights are preserved in modified and extended versions of the work. For MRST, this means that you can use the software for any purpose, share it with anybody, modify it to suit your needs, and share the changes you make. However, if you share any version of the software, modified or unmodified, you must grant others the same rights to distribute and modify it as in the original version. By distributing MRST as free software under the GPLv3 license, the developers of MRST have made sure that the software will stay free, no matter who changes or distributes it.

The development of the MRST toolbox has to a large extent been funded by strategic research grants awarded from the Research Council of Norway. Dissemination of research results is an important evaluation criterion for these types of research grants. To provide the developers with an overview of some usage statistics for the software, you are kindly asked to register your affiliation/country upon download. This information is only used when reporting impact of the creative work to agencies co-funding the development of MRST. If you also leave an email address, we will notify you when a new release or critical bugfixes are available. Your e-mail address will under no circumstances be shared with any third party.

Finally, if you use MRST in any scholar work, we require that the creative work of the MRST developers is courteously and properly acknowledged by referring to the MRST webpage or by citing this book or one of the overview papers that describe MRST [156, 153, 139, 25].

³ <http://homepages.ulb.ac.be/~ynotay/AGMG/>

⁴ See <http://www.gnu.org/licenses/gpl.html> for more details.

COMPUTER EXERCISES:

108. Download and install the software
109. Run `flowSolverTutorial1` from the command line to verify that your installation is working.
110. Load the `flowSolverTutorial1` tutorial in the editor (`edit flowSolverTutorial1.m`) and run it in cell mode (evaluating one cell at the time). Use `help` or `doc` to inspect the documentation for the various functions that are used in the script.
111. Run the `flowSolverTutorial1` tutorial line-by-line: Set a breakpoint on the first executable line by clicking on the small symbol next to line 27, push the 'play button', and then use the 'step' button to advance a single line at the time. Change the grid size to $10 \times 10 \times 25$ and rerun the example.
112. Use `mrstExploreModules()` to locate and load the `incompIntro` tutorial from the `incomp` module. Examine the tutorial in the same way as you did for `flowSolverTutorial1`. Publish the workbook to reproduce the contents of Figure A.3.
113. Replace the constant permeability in the `incompIntro` tutorial by a random permeability field

```
rock.perm = logNormLayers(G.cartDims,[100 10 100])*milli*darcy;
```

Can you explain the changes in the pressure field?

114. Run all of the examples listed by `mrstExamples()` that have the word 'tutorial' in their names. In particular,
 - `gridTutorialIntro` introduces you quickly to the most fundamental parts of MRST, the grid structure, which will be discussed in more detail in Chapter 3;
 - `tutorialPlotting` introduces you to various basic routines and techniques for plotting grids and data defined over these;
 - `tutorialBasicObjects` will give you a quick overview of a lot of the functionality that can be found in the toolbox.

A.2 Public data sets and test cases

Good data sets are in our experience essential to enable tests of new computational methods in a realistic and relevant setting. Such data sets are hard to come by, and when making MRST we have made an effort to provide simple access to a number of public data sets that can be freely downloaded. With the exception of a few illustrations in Chapter 3 that are based on data that cannot be publicly disclosed, all examples discussed in the book either use MRST to create their input data or rely on public data sets that can be downloaded freely from the internet.

To simplify dataset management, MRST offers a graphical user interface,

```
mrstDatasetGUI()
```

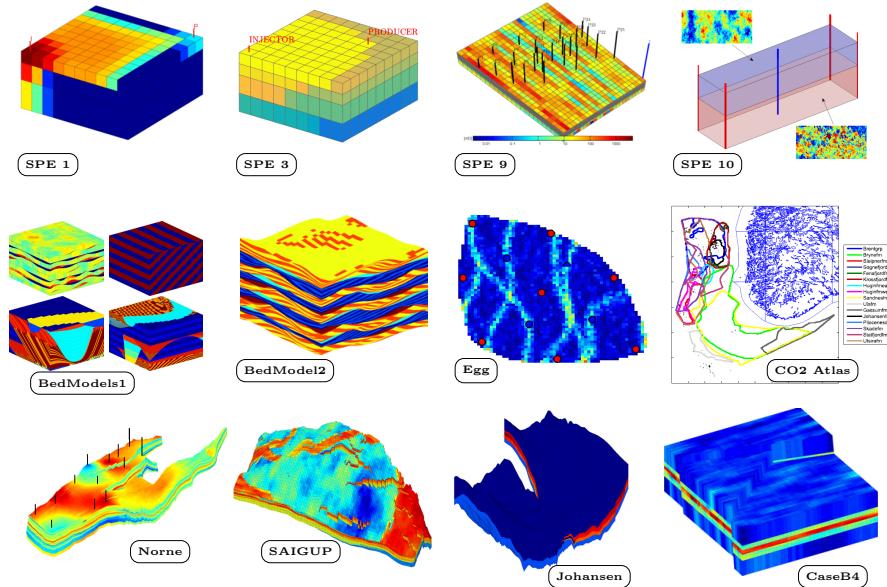


Fig. A.5. Examples of the free data sets that are distributed along with MRST.

that lists all public data sets known MRST, gives a short description of each, and can download and unpack most of the data sets to the correct sub-folder of the standard path. A few data sets require you to register your email address or fill in a license form, and in this case we provide a link to the correct webpage. Figure A.5 shows some of the data sets that are available via the graphical interface. Many of these will be used throughout the book.

Herein, we use the convention that data sets are stored in sub-directories of a standard path, which you can retrieve by issuing the following query

```
mrstDataDirectory()
```

We recommend that you adhere to this convention when using the software as a supplement to the book. If you insist on placing standard data sets elsewhere, we suggest that you use `mrstDataDirectory(<path>)` to reset the default path.

MRST also contains a number of grid factory routines and simplified geo-statistical algorithms you can use to make your own test cases. These will be discussed in more detail in Chapters 3 and 2, respectively. Here, we should add a word of caution about exact reproducibility. Whereas the grid-factory routines in MRST are mostly deterministic and should enable you to create the exact same grids each time you run them (and hence reproduce the test cases discussed in the book), the routines for generating petrophysical data rely on random numbers and will not give the same results in repeated runs. Hence, you can only expect to reproduce plots and numbers that are qualitatively similar whenever these are used unless you make sure to store and reset the seed used by MATLAB's random-number generator.

A.3 More about modules and advanced functionality

As you will recall from Section A.1.1, MRST consist of core functionality and a set of add-on modules that 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 this section, we explain how to load and manage different modules and will also try to explain the basic characteristics of modules, or in other words, the design criteria you can apply if you intend to develop your own module. For completeness, we also provide a brief overview of the comprehensive set of modules that currently are in the official release. Many of these modules will not be discussed at all in the book, which for brevity needs to focus on modules offering functionality of a general interest to a wide audience.

A.3.1 Operating the module system

MRST's module system is a simple facility for extending and modifying the feature set. Specifically, the module system enables on-demand activation and deactivation of advanced features that are not part of the core functionality. The module system consists of two parts; one that handles mapping of module names to system paths and one that uses this mapping to manipulate MATLAB's search path. The search path is a list of folders used by MATLAB to locate files. A module in MRST is strictly speaking a collection of functions, scripts, object declarations, etc., located in a folder. Each module needs to have a unique name and reside in a different folder than other modules. When you activate a module, the corresponding folder is added to the search path, and when you deactivate, the folder is removed.

The mapping between module names and paths on your computer is maintained by the function `mrstPath`. The paths are expected to be full paths to existing folders on your computer. To determine which modules are part of your current installation, you use the function as a command

```
mrstPath
```

This will list all modules that MRST is aware of and can activate. To activate a particular module, you use the function `mrstModule`. As an example, calling

```
mrstModule add mimetic mpfa
```

will load the two modules that contain the consistent discretization methods discussed in Chapter 6. The `mrstModule` function has the following additional modes of operation

```
mrstModule <list|clear|reset|gui> [module list]
```

which will list all active modules, deactivate all modules, deactivate all modules except for those in the list, or bring up a graphical user interface with check-boxes that enable you to activate and deactivate individual modules. For the latter, you can also use the command `moduleGUI`.

All modules that come bundled with the official release will be placed in a predefined folder structure, so that they can be added automatically to the module mapping by the `startup` function. However, module folders can in principle be placed anywhere on your computer as long as they are readable. To make MRST aware of these additional modules, you need to use `mrstPath` to register them in the mapping. Assuming, for instance, that you want to make the AGMG multigrid solver [203] available. This can be done as follows,

```
mrstPath register AGMG S:\mrst\modules\3rdparty\agmg
```

Once the mapping is established, the module can be activated

```
mrstModule add AGMG
```

In my experience, the best way to register modules that are not part of the default mapping that comes with the official versions of MRST is to add appropriate lines to your `startup_user` file. The following is an excerpt from mine

```
mrstPath reregister distmesh ...
/home/kalie/matlab/mrst-bitbucket/mrst-core/utils/3rdparty/distmesh
```

which adds the mesh generator `DistMesh` by Persson and Strang [212] as a module in MRST.

A.3.2 What characterizes a module?

What makes a module in MRST differs quite a lot. Some modules are just small pieces of code that add specialized capabilities like the `mpfa` module, whereas others add comprehensive new functionality such as the `incomp`, `ad-core`, `co2lab` modules, etc. Some of these modules are robust, well-documented, and contain features that will likely not change in future releases. As such, they could have been included as part of the core functionality if we had not decided to keep this as small as possible to simplify maintenance and reduce the potential for feature conflicts. Other modules are constantly changing to support ongoing research. Until recently, all modules in the AD-OO family were of this type. Using the module concept is often convenient if you are working on a project that has specific functionality that you may want to activate or deactivate as you like. In-house, our team uses a large number of modules that are in varying degree of development and/or decay. Some are accessible to the whole team, whereas others reside on one person's computer only. In summary, any module that is part of the official release is just some

code that has been organized in a certain way and released to others. Common for all modules, however, is that they attempt to adhere to the following recommendations:

- A module should offer some new functionality that is distinctly different from what is already available in mrst-core and/or other modules.
- A module should distinguish between functionality that is exposed to the users of the module and functionality that is only used internally. The latter can be put in a folder called `private` (and will then not be accessible to other functions than those in the folder above) or some other folder that signals that these functions are not the main ones to be used
- A module should contain a set of tutorials/examples that explain and highlight basic functionality of the module. If these examples require special data sets, these should be published along with the module.
- All main routines in a module should be documented (similar to what is already done in MRST) describing input and output data, the underlying method, and preferably assumptions and limitations
- Modules should, as a general rule, not use functionality from the official toolboxes that are sold with MATLAB since many users do not have access to these.
- Name conflicts should be avoided to avoid messing up the search path in MATLAB. When two files with the same name appear in multiple folders on the search path, MATLAB will pick the one found nearest the top of the search path. To avoid potential unintended side effects, it is therefore important that files have unique names across different modules.
- To the extent possible, the implementation should try to stick to the same naming conventions as used in MRST for readability. That is use `camelCaseNames` for functions, use `G` for grid, `rock` for petrophysics, `fluid` for fluid models, etc.

A.3.3 List of modules

In this section, we will go through the various modules that make up the official MRST release and explain briefly the purpose and key features of each individual module. To this end, we have followed the subdivision introduced in Section A.1.1, which is slightly different from the one found on the MRST webpages.

Incompressible discretizations and solvers

This family of modules consist primarily of functionality for solving Poisson-type pressure equations:

`incomp`: The module implements the basic fluid objects necessary to represent incompressible, two-phase flow. To solve these flow equations, the module offers a standard two-point flux-approximation (TPFA) method for

pressure and explicit and implicit transport solvers using single-point upstream mobility weighting, which can be combined in sequential solution procedures. The `incomp` module was originally part of the core module, but was moved to a separate module once the software started offering solvers for more advanced fluid models. The module is described in detail in Part II and III of the book.

`mimetic`: The standard two-point flux approximation method implemented in the `incomp` module is not consistent and not hence convergent unless the grid is K-orthogonal. The solver may therefore give significant grid-orientation errors for anisotropic permeabilities and skew grids. The module implements a family of mimetic finite difference methods, which give consistent discretization for incompressible (Poisson-type) pressure equations, see Section 6.3

`mpfa`: The module implements the MPFA-O scheme, which is an example of a multipoint flux-approximation scheme that employs more degrees of freedom when constructing the discrete fluxes across cell interfaces; this to ensure a consistent discretization with reduced grid-orientation effects.

`vem`: Virtual element methods (VEM) [34, 35] constitute a unified framework for higher-order methods on general polygonal and polyhedral grids. The module can be used to solve general incompressible flow problems using first- and second-order VEM, with the possibility to choose different inner products. The module was originally developed by Klemetsdal as part of his master thesis [138].

Implicit solvers based on the AD-OO framework

Although the framework offers functionality for solving a wide class of model equations, the current functionality is primarily geared towards compressible multiphase flow.

`ad-core`: Object-oriented framework for solvers based on automatic differentiation (MRST AD-OO). This module by itself does not contain any complete simulators, but rather implements the common framework used for many other modules. This framework includes abstract classes for reservoir models, for time stepping, nonlinear solvers, linear solvers, functionality for variable updating/switching, routines for plotting well responses, etc. See the discussion in Chapter 11 and [139, 25] for more details.

`ad-blackoil`: Models and examples that extend the MRST AD-OO framework found in the `ad-core` module to black-oil problems. More specifically, the module adds additional mathematical models that implement the black-oil equations for multiphase, miscible, compressible flow and offers models for single-phase, two-phase and three-phase flow. The three-phase models include dissolution of gas into oil and/or oil vaporizing into the gas phase. The solvers support source terms, boundary conditions and complex wells with changing controls, production limits and multiple segments. The module includes a wide variety of examples validating the

solvers against a commercial simulator on standard test cases (SPE 1 and SPE 9). See Chapter 11 for more details.

blackoil-sequential: This module implements sequential solvers for the same set of equations that are implemented using a fully-implicit discretization in `ad-blackoil`. The solvers are based on a fractional flow formulation wherein pressure and transport are solved as separate steps, see [182] for more details. These solvers can be significantly faster than those found in `ad-blackoil` for many problems, especially problems where the total velocity changes slowly during the simulation.

ad-eor: This module builds on top of the AD-OO framework from the `ad-core` and `ad-blackoil` modules, and defines model equations and provides simulation tools for water-based enhanced oil recovery techniques (polymer and surfactant injection), see [25] for more details about the polymer case.

optimization: The module contains routines for solving optimal control problems with (forward and adjoint) solvers based on the AD-OO framework. The module contains a quasi-Newton optimization routine using BFGS-updated Hessians, but can easily be set up to use any (non-MRST) optimization code.

ad-props: Functionality related to property calculations for the AD-OO framework. Specifically, the module implements a variety of test fluids and functions that are used to create fluids from external datasets. This module is used as part of almost all simulators in MRST studying compressible equations of black-oil or compositional type.

Workflow tools

The two families of modules outlined above are concerned with simulation of a given reservoir model. The next category of modules contain functionality that can be used either before or after a reservoir simulation as part of a general modeling workflow.

diagnostics: Flow diagnostics tools are run to establish connections and basic volume estimates, and quickly provide a qualitative picture of the flow patterns in the reservoir. These methods offer a computationally inexpensive alternative to multiphase simulations to rank, compare, and validate reservoir models, upscaling procedures, and production scenarios. The flow diagnostics module contains utilities for computing time-of-flight, tracer partitioning, and measures of dynamic heterogeneity, which in turn can be used to compute well-allocation factors, drainage/sweep regions, simple estimates of recovery and net-present-value, etc. See Chapter 13 and [184, 220, 141] for more details.

upr: The module contains functionality for generating unstructured polyhedral grids that align to prescribed geometric objects. Control-point alignment of cell centroids is introduced to accurately represent horizontal and multilateral wells, but can also be used to create volumetric representations of fracture networks. Boundary alignment of cell faces is introduced

to accurately preserve geological features such as layers, fractures, faults, and/or pinchouts. This third-party module is developed based on a master thesis by Berge [38], see also [206].

coarsegrid: The module contains extended functionality for defining coarse grids formed as a partition of an underlying fine grid. Such grids form a key part in upscaling and multiscale methods, but act almost like any standard MRST grid and can hence be passed to most of the solvers from other modules. Coarse grid generated from a partition are also useful for visualization purposes. More details are given in Chapter 14.

agglom: The module contains a set of tools and tutorials for constructing coarse grid based upon amalgamation of cells from a fine-scale model [102, 101, 158, 162]. Primary examples are coarse grids that adapt to geological features such as flow units, deposition environment, and various types of regions (facies, stone types, initialization regions, etc). Likewise, using an amalgamation process that consists of a sequence of merging and splitting operations, one can generate flow-adapted grids based on an indicator like flux magnitude, time-of-flight, etc.

upscaling: The module includes methods for flow-based upscaling of both permeabilities and transmissibilities on general grids, as well as a few tutorials demonstrating how to implement simple averaging methods. See Chapter 15 for more details.

steady-state: Functionality for upscaling relative permeabilities based on a steady-state assumption. This includes general steady state, as well as capillary and viscous-dominated limits. The functionality is demonstrated through a few tutorial examples. For more details, see Hilden [106, 105].

adjoint: The module implements strategies for production optimisation based on adjoint formulations. This enables, for instance, optimization of the net present value constrained by the bottom-hole pressure in wells. This module is limited to two-phase, incompressible flow as implemented in the `incomp` module. For optimization problems with more complex fluid physics, the newer `optimization` module from the AD-OO framework is recommended.

enkf: An ensemble Kalman filter (EnKF) module developed by researchers at TNO [149, 148] that contains EnKF and EnRML schemes, localization, inflation, asynchronous data, production and seismic data, updating of conventional and structural parameters.

remso: The REservoir Multiple Shooting Optimizer (REMSO) is an optimization module based on multiple shooting, developed by Codas [74]. It allows for great flexibility in the handling of non-linear constraints in reservoir management optimization problems.

Multiscale methods

Multiscale methods can either be used as a robust upscaling method to produce upscaled flow velocities and pressures on a coarse grid or as an approximate, iterative fine-scale solver. Instead of placing these methods into one of

the categories above, we have given them a separate category, in part because of the prominent place they have played as a driving force for the development of MRST:

`msmfe`: The module implements the multiscale mixed finite-element (MsMFE) method on stratigraphic and unstructured grids in 3D [59, 1, 2, 3, 5, 135, 188, 13, 158, 207]. The idea is to introduce a set of basis functions (prolongation operators) associated with the faces of a coarse grid. The basis functions are computed numerically by solving localized flow problems on pairs of coarse blocks. Using these basis functions, one can define a reduced flow problem on the coarse grid and prolongate the resulting coarse solution back to the fine grid to get a mass-conservative flux field. The method is a good approximative solver, and a robust and accurate alternative to upscaling, for incompressible flow problems. MRST was originally developed for the sole purpose of supporting the research on MsMFE methods. This module has not been actively developed since 2012.

`msfv`: The module implements the operator formulation [166] of the multiscale finite-volume (MsFV) method [118] for incompressible flow on structured and unstructured grids in 3D, under certain restrictions on the grid geometry and topology [175, 177]. The key idea is to introduce a set of prolongation operators, defined by solving flow problems localized to dual coarse blocks. The prolongation operators are then used to define a reduced flow problem on the coarse primal grid and map unknowns computed on this grid back to the underlying fine grid. The resulting method can either be used as an approximate solver or as a global preconditioner in an iterative solver. The module is not actively developed and is offered mostly for historic reasons. Users interested in applying multiscale methods should look at the `msrsb` module instead.

`msrsb`: The Multiscale Restricted-Smoothing Basis (MsRSB) method [181, 182] is the current state-of-the-art within multiscale methods [161]. MsRSB is very robust and versatile can either be used as an approximate coarse-scale solver having mass-conservative subscale resolution, or as an iterative fine-scale solver that will provide mass-conservative solutions for any given tolerance. The performance of the method has been demonstrated on incompressible 2-phase flow [181], compressible 2 and 3-phase-black oil models [182], as well as compositional models [183]. It has also been demonstrated that the method can utilize combinations of multiple prolongation operators [163], e.g., corresponding to coarse grids with different resolutions, adapting to geological features, adapting to wells, or moving displacement fronts. At the moment, only parts of this functionality is available in the public version of the module.

Specialized simulation tools

This category consists of modules that implement solvers and simulators for other mathematical models than those discussed in this book.

co2lab: This module combines results of more than one decade of academic research and development on CO₂ storage modeling into a unified toolchain that is easy and intuitive to use. The module is geared towards the study of long-term trapping in large-scale saline aquifers and offers computational methods and graphical user interfaces that enable you to visualize migration paths and compute upper theoretical bounds on structural, residual, and solubility trapping. The module also offers efficient simulators based on a vertical-equilibrium (VE) formulation that can be used to analyse pressure build-up and plume migration and compute detailed trapping inventories for realistic storage scenarios. Last but not least, the module provides simplified access to publicly available data sets, e.g., from the Norwegian CO₂ Storage Atlas. For more details, see the following references: the general tool chain of methods [197, 159, 17, 160, 18], structural traps [198], and VE formulations [16, 195, 196].

dfm: The module contains two-point and multipoint solvers for discrete fracture-matrix systems [225, 248], including multiscale methods. This third-party module is developed by Sandve from the University of Bergen, with minor modifications by Keilegavlen.

hfm: The hierarchical fracture module (HFM) utilises the hierarchical fracture modelling framework to simulate multiphase flow in naturally fractured reservoirs with multiple length scales. Also known as the embedded discrete fracture model, this method models fractures explicitly, as major fluid pathways, and benefits from independent definitions of the fracture and matrix grid. As a result, intricate fracture networks can be modelled easily, without the need for a complex underlying matrix grid that is conformal with each fracture. The module also extends the newly developed multiscale restriction smoothed basis (MsRSB) method to compute the flux field developed in a fractured reservoir, see [229] for more details.

dual-porosity: A module for geologic well-testing in naturally fractured reservoirs has been developed by the Carbonate Reservoir Group at Heriot Watt university. The module setups tools to generate synthetic transient pressure responses for idealized and realistic fracture networks.

vemmech: This module offers functionality to set up solvers for linear elasticity problems on irregular grid, using the virtual element method [35, 92], which is a generalization of finite-element methods that takes inspiration from modern mimetic finite-difference methods. For more details, see [19, 199].

fvbiot: The module, developed by Nordbotten and Keilegavlen, implements cell-centered discretizations of three different equations:

- Scalar elliptic equations (Darcy flow), using multipoint flux approximations (this is more or less equivalent to the MPFA implementation in the `mpfa` module, although the implementation and data structures are slightly different).
- Linear elasticity, using the multipoint stress-approximation (MPSA) method [201, 202, 128].

- Poro-mechanics, e.g., coupling terms for the combined system of the above two models.

Utility routines

The last category consists of modules that do not offer any computational or modeling tools but rather provide general utility functions that can be utilized by the other modules in MRST:

`mrst-gui`: Graphical interfaces for interactive visualization of reservoir states and petrophysical data. The module includes additional routines for improved visualization (histograms, well data, etc) as well as a few utility functions that enable you to override some of MATLAB's settings to enable faster 3D visualization (rotation, etc).

`book`: Module containing all the scripts used for the examples, figures, and some of the exercises in this book.

`deckformat`: The module contains support for input of complete simulation decks in the ECLIPSE format, including input reading, conversion to SI units, and construction of MRST objects for grids, fluids, rock properties, and wells. Functionality from this module is thus essential for the fully implicit simulators developed in the `ad-blackoil` and `ad-eor` modules, but is also used in a large number of MRST's examples and tutorials as well as many of the examples herein.

`spe10`: Model 2 from the 10th SPE Comparative Solution Project has developed into a standard test case used by a large number of researchers. The `spe10` module contains tools for downloading, converting, and loading the data into MRST. The module also features utility routines for extracting parts of the model, as well as a script that sets up a (crude) simulation of the full model (using the AGMG multigrid solver). For more details, see Section 2.5.3.

`libgeometry`: Many of the routines in MRST rely on detailed geometric information about the grids including cell volumes and centroids, face areas and normals, etc. Computing this information efficiently in MATLAB is not an easy task and for very large and complex grids, the builtin functionality from the core module can be too slow. The current module therefore offers C-accelerated computation of geometric grid properties (cell centroids and volumes and face centroids, areas, and normal vectors).

`opm_gridprocessing`: Although the `computeGeometry` function from MRST's core functionality is generally efficient, it can be slow for large models. To accelerate the processing of large models, this module offers access to grid-processing routines written in C for the Open Porous Media (OPM) initiative, which generally can be seen as the C/C++ siebling of MRST.

`streamlines`: The module implements Pollock's method for tracing of streamlines on Cartesian and curvilinear grids based on a set of input fluxes, as computed by the incompressible flow solvers in MRST (i.e., from the `incomp`, `mimetic`, or `mpfa` module).

wellpaths: Functionality for defining wells following curvilinear trajectories.

As should be evident from the overview of current modules, MRST does not have strict requirements on what becomes a module and what does not. The concept of semi-independent modules is simply a way to organize the software development that promotes software reuse. If you start to make what should eventually become a module, you will probably be a bit more careful to distinguish parts of your development that have generic value from parts that are case specific or of temporary value only. Moreover, the fact that others or your future self may want to reuse your functionality will hopefully also motivate you to put in the extra effort to document your routines and make examples and tutorials that later decided whether somebody wants to use or continue to develop the functionality you have implemented or not.

COMPUTER EXERCISES:

115. Try to run the following tutorials and examples from various modules
- `simpleBCmimetic` from the `mimetic` module.
 - `simpleUpscaleExample` from the `upscale` module
 - `gravityColumnMS` from the `mssfem` module
 - `example2` from the `diagnostics` module
 - `firstTrappingExample` from the `co2lab` module (notice that this example does not work unless you have **MATLAB-BGL** installed).

A.4 Rapid prototyping using MATLAB and MRST

How can you reduce the time span from the moment you get a new idea to when you have demonstrated that it works well for realistic reservoir engineering problems?

In our experience, prototyping and validating new numerical methods is painstakingly slow. There are many reasons for this. First of all, there is often a strong disconnect between the mathematical abstractions and equations used to express models and numerical algorithms and the syntax of the computer language you use to implement your algorithms. This is particularly true for compiled languages, where you typically end up spending most of your time writing and tweaking loops that perform operations on individual members of arrays or matrices. Object-oriented languages like C++ offer powerful functionality that can be used to make abstractions that are both flexible and computationally efficient and enable you to design your algorithms using high-level mathematical constructs. However, these advanced features are usually alien and unintuitive to those who do not have extensive training in computer sciences. If you are familiar with such concepts and are in the possession of a flexible framework, you still face the never-ending frustration

caused by different versions of compilers and (third-party) libraries that seems to be an integral part of working with compiled languages.

Based on the experience of a large number of researchers and students over the past twenty years, we claim that using a numerical computing environment based on a scripting language like MATLAB (or Python) to prototype, test, and verify new models and computational algorithms is significantly more efficient than using a compiled language like Fortran, C, and C++. Not only is the syntax intuitive and simple, but there are many mechanisms that help to boost your productivity and you avoid some of the frustrations that come with compiled languages: there is no complicated build process or handling of external libraries and your implementation is inherently cross-platform compatible.

MATLAB, for instance, provides mathematical abstractions for vectors and matrices and built-in functions for numerical computations, data analysis, and visualization that enable you to quickly write codes that are not only compact and readable, but also efficient and robust. On top of this, MRST provides additional functionality that has been developed especially for computational modeling of flow in porous media:

- an unstructured grid format that enables you to implement algorithms without knowing the specifics of the grid;
- discrete operators, mappings, and forms that are not tied to specific flow equations, and hence can be precomputed independently and used to write discretized flow equations in a very compact form that is close to the mathematical formulations of the same;
- automatic differentiation which enables you to compute the values of gradients, Jacobians, and sensitivities of any programmed function without having to compute the necessary partial derivatives analytically; this can, in particular, be used to automate the formulation of fully implicit discretizations of time-dependent and coupled systems of equations
- data structures that provide unified access and enable you to hide specific details of constitutive laws, fluid and rock properties, boundary conditions, wells, etc;

This functionality will be gradually introduced and described in detail throughout the book.

An equally important aspect of using a numerical environment like MATLAB is that you can develop your program differently than what you would do in a compiled language. Using the interactive environment, you can interactively analyse, change and extend data objects, try out each operation, include new functionality, and build your program as you go. This feature is essential in a debugging process, when one tries to understand why a given numerical method fails to produce the results one expects it to give. In fact, you can easily continue to change and extend your program during a test run: the debugger enables you to run the code line by line and inspect and change variables at any point. You can also easily step back and rerun parts of the

code with changed parameters that may possibly change the program flow. Since MATLAB uses dynamic type-checking you can also add new behavior and data members while executing a program. However, how to do this in practice is difficult to teach in a textbook. Instead, you should run and modify the various examples that come with MRST and the book. We also recommend that you try to solve the computer exercises that are suggested at the end of several of the chapters and sections in the book.

Unfortunately, all this flexibility and productivity comes at a price: it is very easy to develop programs that are not very efficient. In the book, we therefore try to teach programming concepts that can be used to ensure flexibility and high efficiency of your programs. These include, in particular, powerful mechanisms for traversing data structures like vectorization, indirection maps, and logical indexes, as well as use of advanced MATLAB functions like `accumarray`, `bsxfun`, etc. Although these will be presented in the context of reservoir simulation, we think the techniques should be of interest for readers working with lower-order finite-volume discretizations on general polyhedral grids. As an illustration of the type of MATLAB programming that will be used, let us consider a simple example. The following code generates one million random points in 3D and counts the number of points that lie inside each of the eight octants:

```
n = 1000000;
pt = randn(n,3);
I = sum(bsxfun(@times, pt>0, [1 2 4]),2)+1;
num = accumarray(I,1);
```

The third line computes the sign of the x , y , and z coordinates and maps each of the resulting one million triples of logical values to an integer number between 1 and 8 that represents each of the octants. To count the number of points inside each octant, we use the function `accumarray` that groups elements from a data set and applies a function to each group. The default function is summation, and by setting a unit value in each element, we count the entries.

Next, let us compute the mean point inside each octant. A simple loop-based solution could be something like:

```
avg = zeros(8,3);
for i=1:1000000
    quad = sum((pt(i,:)>0).*[1 2 4])+1;
    avg(quad,:) = avg(quad,:)+pt(i,:);
end
avg = bsxfun(@rdivide, avg, num);
```

Use of loops should generally be avoided since they tend to be slow in MATLAB. On the author's computer, it took 0.09 seconds to count the number of points within each octant, but 2.6 seconds to compute the mean points. So let us try to do something more clever and utilize vectorization. The `accumarray` function cannot be used since it only works for scalar values. Instead, we can

build a sparse matrix that we multiply with the `pt` array to sum the coordinates of the points. The matrix will have one row per octant and one column per point. Now, all we have to do is to use our indicator `I` to assign a unit value in the correct row for each column use `bsxfun` to divide the sum of the coordinates with the number of points inside each octant:

```
avg = bsxfun(@rdivide, sparse(I,1:n,1)*pt, num);
```

On the author's computer this operation took 0.05 seconds, which is fifty times faster than the loop-based solution. In fact, summing all coordinates inside each octant is faster than counting the number of points. Let us try to utilize this to speed up the computation of mean points. This is quite simple: we expand each coordinate to a quadruple $(x, y, z, 1)$, multiply by the same sparse matrix, and use `bsxfun` to divide the first three columns by the fourth column to compute the average:

```
avg = sparse(I,1:n,1)*[pt, ones(n,1)];
avg = bsxfun(@rdivide, avg(:,1:end-1), avg(:,end));
```

The overall operation ran in 0.07 seconds, which not only is two times faster than our previous solution, but perhaps also a bit more elegant.

Hopefully, this simple example has inspired you to learn a bit more about efficient programming tricks if you do not already speak MATLAB fluently. MRST is generally full of tricks like this, and in the book we will occasionally show a few of them. However, if you really want to learn the tricks of the trade, the best way is to dig deep into the actual codes.

A.5 Automatic differentiation in MRST

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 1 is the derivative dx/dx and f_x is the numerical value of the derivative df/dx . Accordingly, the action of the elementary operations and functions must be defined for such pairs, e.g.,

$$\begin{aligned}\langle f, f_x \rangle + \langle g, g_x \rangle &= \langle f + g, f_x + g_x \rangle, \\ \langle f, f_x \rangle * \langle g, g_x \rangle &= \langle fg, fg_x + f_xg \rangle, \\ \langle f, f_x \rangle / \langle g, g_x \rangle &= \langle f/g, (f_xg - fg_x)/g^2 \rangle \\ \exp(\langle f, f_x \rangle) &= \langle \exp(f), \exp(f)f_x \rangle,\end{aligned}$$

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, these rules should not be implemented as standard functions, so that you need to write something like `myPlus(a, myTimes(b,c))` when you want to evaluate $a + bc$. An elegant solution is to instead use classes and operator overloading. When MATLAB encounters an expression `a+b`, the software will choose one out of several different addition functions depending on the data types of `a` and `b`. All we now have to do is introduce new addition functions for the various classes of data types that `a` and `b` may belong to. Neidinger [190] gives a nice introduction to how to implement this in MATLAB. .

There are many automatic differentiation libraries for MATLAB, e.g., ADi-Mat [239, 40], ADMAT [51, 250], MAD [243, 231, 90], or from MATLAB Central [88, 173]. The AD class in MRST relies on operator overloading as suggested in [190] 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 that 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. In typical cases, we will use fully-implicit discretizations in which one seeks to solve for all state variables simultaneously, but we may still want to manipulate parts of the full equation system that 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 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 as a list of matrices.

In the rest of the section, we will go through a few relatively simple examples that demonstrate how the AD class works. Later in the book we demonstrate how automatic differentiation can be used to set up simulations in a (surprisingly) few number of code lines.

Example A.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 $y = 2$. Using our previous notation, AD-representation of z should be an

object of the following form

$$z = \langle 3e^{-xy}, -3ye^{-xy}, -3xe^{-xy} \rangle \approx \langle 0.4060, -0.8120, -0.4060 \rangle.$$

Computing z and its partial derivatives 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 initializes 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):

$x = \text{ADI Properties:}$	$y = \text{ADI Properties:}$	$z = \text{ADI Properties:}$
val: 1	val: 2	val: 0.4060
jac: {[1] [0]}	jac: {[0] [1]}	jac: {[[-0.8120] [-0.4060]}}

$\frac{\partial x}{\partial x}$	$\frac{\partial x}{\partial y}$	$\frac{\partial y}{\partial x}$	$\frac{\partial y}{\partial y}$	$\frac{\partial z}{\partial x} \Big _{x=1,y=2}$	$\frac{\partial z}{\partial y} \Big _{x=1,y=2}$
---------------------------------	---------------------------------	---------------------------------	---------------------------------	---	---

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 happens 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 A.6 gives an overview of the sequence of calls that are invoked within the AD implementation to evaluate $3*\exp(-x*y)$ when x and y are AD variables⁵.

⁵ The observant reader may notice that some computational saving could have been obtained 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.

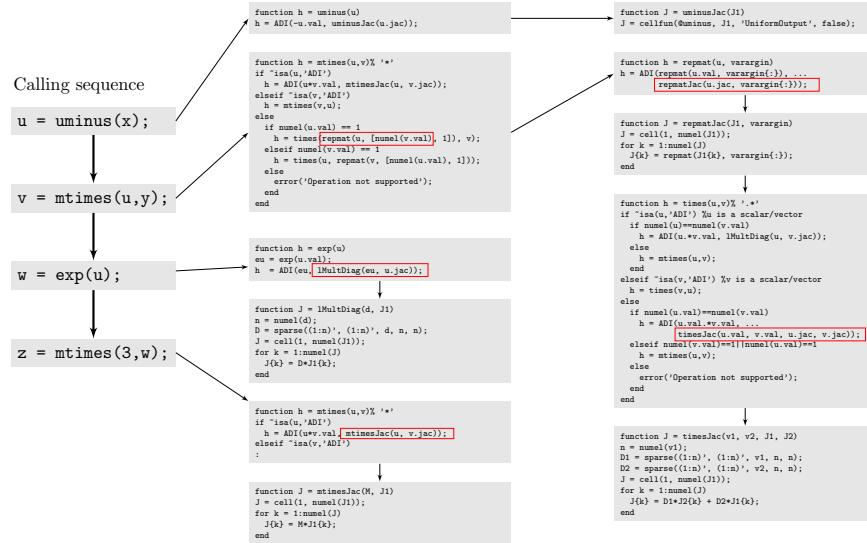


Fig. A.6. Complete set of functions invoked to evaluate $3 \cdot \exp(-x \cdot y)$ when x and y are AD variables. For brevity, we have not included details of the constructor function $\text{ADI}(\text{val}, \text{Jac})$, which constructs an AD pair with the value val and list of Jacobi matrices Jac .

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 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 A.2. To investigate the efficiency of vectorization versus serial execution of the AD objects in MRST, we consider the inner product of two vectors

$$z = x \cdot y;$$

We will compare the cost of computing z , $\partial z / \partial x$, and $\partial z / \partial y$ using four different approaches:

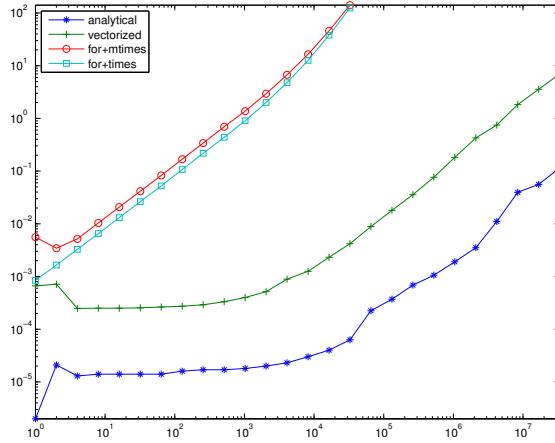


Fig. A.7. Comparison of the time required for computing $z=x.*y$ and derivatives as function of the number of elements in the vectors x and y .

1. analytical expressions $z_x = y$ and $z_y = x$ and standard MATLAB vectors of doubles,
2. the overloaded vector multiply ($.*$) with AD-vectors for x and y
3. a loop over all vector elements with matrix multiply ($*=mtimes$) and x and y represented as scalar AD variables
4. same as 3, but with vector multiply ($*=times$)

This is implemented as follows:

```
[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 A.7 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 a loop will on average be more than three orders more expensive than using vectorization. Since the inner iterations multiplies scalars, many programmers would implement it using matrix multiply $*$ without a second thought. Replacing $*$ by vector multiply $.*$ reduces the cost by 30% on average, but the factor diminishes as the number of elements increases in the vector.

While **for**-loops in many cases will be quite efficient in MATLAB (contrary to common belief), one should always try to avoid loops that call functions that have 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.

The main use of AD objects in MRST is to linearize and assemble (large) systems of discrete equations. To use AD to assemble and solve a linear system $\mathbf{Ax} = \mathbf{b}$, we must first write the system on residual form,

$$\mathbf{r}(\mathbf{x}) = \mathbf{Ax} - \mathbf{b} = \mathbf{0}. \quad (\text{A.1})$$

Since \mathbf{x} is unknown, we will assume that we have an initial guess called \mathbf{y} . Inserting this into (A.1), we obtain

$$\mathbf{r}(\mathbf{y}) = \mathbf{Ay} - \mathbf{b} = \mathbf{A}(\mathbf{y} - \mathbf{x}),$$

which we can solve for \mathbf{x} to obtain $\mathbf{x} = \mathbf{y} - \mathbf{A}^{-1}\mathbf{r}(\mathbf{y})$. It follows from (A.1) that $\partial\mathbf{r}/\partial\mathbf{x} = \mathbf{A}$, which means that if we can write a code that evaluates the residual for each and every equation that makes up our system, we can use automatic differentiation to assemble and solve the system.

Example A.3. As a very simple illustration of how automatic differentiation can be used to assemble a linear system, let us consider the following linear 3×3 system

$$\begin{bmatrix} 3 & 2 & -4 \\ 1 & -1 & 2 \\ -2 & -2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -5 \\ -1 \\ 6 \end{bmatrix},$$

whose solution $x = [1 \ 2 \ 3]^T$ can be computed by a single line in MATLAB:

```
x = [3, 2, -4; 1, -4, 2; -2, -2, 4]\ [-5; -1; 6]
```

To solve the same system using automatic differentiation, we need the following lines:

```
x = initVariablesADI(zeros(3,1));
eq1 = [ 3, 2, -4]*x + 5;
eq2 = [ 1, -4, 2]*x + 1;
eq3 = [-2, -2, 4]*x - 6;
eq = cat(eq1,eq2,eq3);
u = -eq.jac{1}\eq.val
```

Here, the first line sets up the AD variable x and initializes it to zero. The next three lines evaluate the equations that make up our linear system. Evaluating each equation results in a scalar residual value $eq1.val$ and a 1×3 Jacobian matrix $eq1.jac$. In the fifth line, the residuals are concatenated to form a vector and the Jacobians are assembled into the full Jacobian matrix of the overall system.

At this point, you may argue that what I have shown you is a convoluted and expensive way of setting up and solving a simple system. However, now comes the interesting part. When solving partial differential equations on complex grids, it is often much simpler to evaluate the residual equations in each grid cell than assembling the same local equations into a global system matrix. Using automatic differentiation, you can focus on the former and avoid the latter. In Chapter 4.4.2, we will introduce discrete divergence and gradient operators, `div` and `grad`. With these, the evaluation and assembly of the discrete flow model introduced in Section 1.4 on page 18 can be written in a form that strongly resembles its continuous form

```
eq = div((T/mu)*(grad(p) - g*rho*grad(z)));
```

where `T` is the transmissibilities (which can be precomputed for a given grid), `mu` and `rho` are constant fluid viscosity and density, `g` is the gravity constant, and `z` is the vector of cell centroids, which can be extracted from the grid structure as `z=G.cell.centroids(:,3)`.

Implicitly assembly through automatic differentiation is an alternative to the explicit, procedural approach implemented in the incompressible solver `incompTPFA` introduced in Section 1.4. We will come back to this way of implementing flow models in Section 4.4.2. However, our primary use of automatic differentiation is for compressible flow models, which typically give large systems of nonlinear discrete equations that need to be linearized and solved using a Newton–Raphson method. As a precursor to the discussion in Chapter 7, we will in the last example show how you can use AD to solve a system of nonlinear equations.

Example A.4. Minimizing the Rosenbrock equation

$$f(x, y) = (a - x)^2 + b(y - x^2)^2 \quad (\text{A.2})$$

is a classical test problem from optimization. This problem is often called Rosenbrock's valley or banana function since the global minimum (a, a^2) is located inside a long, narrow and relatively flat valley of parabolic shape. While finding this valley is straightforward, it is more challenging to converge to the global minimum. A necessary condition for a global minimum is that $\nabla f(x, y) = 0$, which translates to the following two equations for (A.2)

$$\mathbf{g}(\mathbf{x}) = \begin{bmatrix} \partial_x f(x, y) \\ \partial_y f(x, y) \end{bmatrix} \begin{bmatrix} -2(a - x) - 4bx(y - x^2) \\ 2b(y - x^2) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \quad (\text{A.3})$$

To derive the Newton–Raphson method, assume that we have a guess \mathbf{x} and want to move to the correct solution $\mathbf{x} + \Delta\mathbf{x}$. Then we can solve for $\Delta\mathbf{x}$ from the following equation

$$\mathbf{0} = \mathbf{g}(\mathbf{x} + \Delta\mathbf{x}) \approx \mathbf{g}(\mathbf{x}) + \nabla\mathbf{g}(\mathbf{x})\Delta\mathbf{x} \quad (\text{A.4})$$

This is quite simple using AD in MRST:

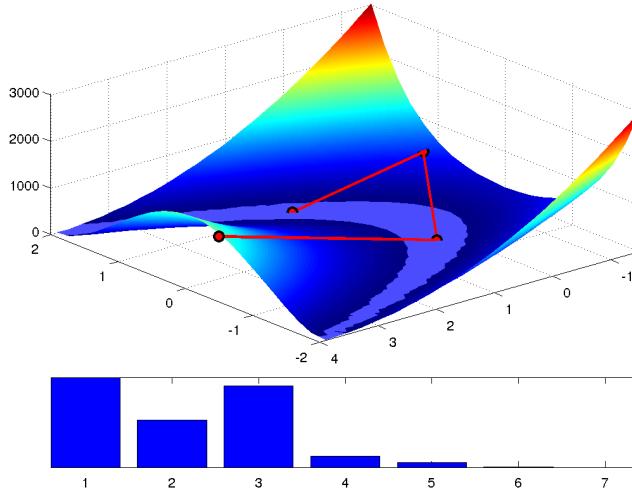


Fig. A.8. Convergence of the AD-based Newton solver for the Rosenbrock problem. The upper plot shows the path taken by the nonlinear solver superimposed over the function $f(x,y)$. Here, we have used a modified colormap to show the valley in which the function f attains its 1% lowest values. The lower plot depicts $f(x,y)^{1/10}$ for the initial guess and the six iterations needed to reduce the norm of the increment below 10^{-6} .

```
[a,b,tol] = deal(1,100,1e-6);
tol = 1e-6;
[x0,incr] = deal([-5;4]);
while norm(incr)>tol
    x = initVariablesADI(x0);
    eq = cat( 2*(a-x(1)) - 4*b.*x(1).*(x(2)-x(1).^2), ...
              2*b.*x(2)-x(1).^2);
    incr = - eq.jac{1}\eq.val;
    x0 = x0 + incr;
end
```

This is just a backbone version of a Newton solver that e.g., does not contain safeguards of any kind like checking that the increments are finite, ensuring that the loop terminates after a finite number of iterations, etc. Figure A.8 illustrates how the Newton solver converges to the global minimum.

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.

COMPUTER EXERCISES:

116. As an alternative to using automatic differentiation, one can use finite differences, $f'(x) \approx [f(x + h) - f(x)]/h$, or a complex extension to compute $f'(x) \approx \text{Im}(f(x + ih))/h$. Use automatic differentiation and the function

```
f = @(x) exp((x-.05).*(x-.4).*(x-.5).*(x-.7).*(x-.95));
```

to assess how accurate the two methods approximate $f'(x)$ at n equidistant points in the interval $x \in [0, 1]$ for different values of h .

117. While the AD class supports `log` and `exp`, it does not yet support `log2`, `log10`, and `logm`. Study `ADI.m` and see if you can implement the missing functions. What about trigonometric functions?
118. Can automatic differentiation be used to compute higher-order derivatives? How or why not?

References

- [1] J. E. Aarnes. On the use of a mixed multiscale finite element method for greater flexibility and increased speed or improved accuracy in reservoir simulation. *Multiscale Model. Simul.*, 2(3):421–439, 2004. ISSN 1540-3459. doi:[10.1137/030600655](https://doi.org/10.1137/030600655).
- [2] 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](https://doi.org/10.1016/j.advwatres.2004.10.007).
- [3] J. E. Aarnes, S. Krogstad, and K.-A. Lie. A hierarchical multiscale method for two-phase flow based upon mixed finite elements and nonuniform coarse grids. *Multiscale Model. Simul.*, 5(2):337–363, 2006. ISSN 1540-3459. doi:[10.1137/050634566](https://doi.org/10.1137/050634566).
- [4] 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, Berlin Heidelberg New York, 2007. doi:[10.1007/978-3-540-68783-2_9](https://doi.org/10.1007/978-3-540-68783-2_9).
- [5] 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](https://doi.org/10.1007/s10596-007-9072-8).
- [6] I. Aavatsmark. An introduction to multipoint flux approximations for quadrilateral grids. *Comput. Geosci.*, 6:405–432, 2002. doi:[10.1023/A:1021291114475](https://doi.org/10.1023/A:1021291114475).
- [7] 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.
- [8] 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.
- [9] J. H. Abou-Kassem, S. M. Farouq-Ali, and M. R. Islam. *Petroleum Reservoir Simulations*. Elsevier, 2013.

- [10] AGMG. Iterative solution with AGgregation-based algebraic MultiGrid, 2012. <http://homepages.ulb.ac.be/~ynotay/AGMG/>.
- [11] 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.
- [12] M. B. Allen, G. A. Behie, and J. A. Trangenstein. *Multiphase flow in porous media: mechanics, mathematics, and numerics*. Lecture notes in engineering. Springer-Verlag, 1988. ISBN 9783540967316.
- [13] F. O. Alpak, M. Pal, and K.-A. Lie. A multiscale method for modeling flow in stratigraphically complex reservoirs. *SPE J.*, 17(4):1056–1070, 2012. doi:[10.2118/140403-PA](https://doi.org/10.2118/140403-PA).
- [14] 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.
- [15] O. Andersen. *Simplified models for numerical simulation of geological CO₂ storage*. PhD thesis, University of Bergen, 2017. URL <http://hdl.handle.net/1956/15477>.
- [16] O. Andersen, S. Gasda, and H. Nilsen. Vertically averaged equations with variable density for CO₂ flow in porous media. *Transp. Porous Media*, pages 1–33, 2014. ISSN 0169-3913. doi:[10.1007/s11242-014-0427-z](https://doi.org/10.1007/s11242-014-0427-z).
- [17] O. Andersen, H. M. Nilsen, and K.-A. Lie. Reexamining CO₂ storage capacity and utilization of the Utsira Formation. In *ECMOR XIV – 14th European Conference on the Mathematics of Oil Recovery, Catania, Sicily, Italy, 8–11 September 2014*. EAGE, 2014. doi:[10.3997/2214-4609.20141809](https://doi.org/10.3997/2214-4609.20141809).
- [18] O. Andersen, K.-A. Lie, and H. M. Nilsen. An open-source toolchain for simulation and optimization of aquifer-wide CO₂ storage. *Energy Procedia*, :1–10, 2015. The 8th Trondheim Conference on Capture, Transport and Storage.
- [19] O. Andersen, H. M. Nilsen, and X. Raynaud. On the use of the virtual element method for geomechanics on reservoir grids. *arXiv preprint arXiv:1606.09508*, 2016.
- [20] J. R. Appleyard and I. M. Cheshire. Nested factorization. In *SPE Reservoir Simulation Symposium, 15–18 November, San Francisco, California*, 1983. doi:[10.2118/12264-MS](https://doi.org/10.2118/12264-MS).
- [21] T. Arbogast, L. C. Cowsar, M. F. Wheeler, and I. Yotov. Mixed finite element methods on nonmatching multiblock grids. *SIAM J. Num. Anal.*, 37(4):1295–1315, 2000.
- [22] H. Ates, A. Bahar, S. El-Abd, M. Charfeddine, M. Kelkar, and A. Datta-Gupta. Ranking and upscaling of geostatistical reservoir models using streamline simulation: A field case study. *SPE Res. Eval. Eng.*, 8(1):22–32, 2005. doi:[10.2118/81497-PA](https://doi.org/10.2118/81497-PA).

- [23] K. Aziz and A. Settari. *Petroleum Reservoir Simulation*. Elsevier Applied Science Publishers, London and New York, 1979.
- [24] L. E. Baker. Three-phase relative permeability correlations. In *SPE Enhanced Oil Recovery Symposium, Tulsa, Oklahoma, 16-21 April, 1988*. doi:[10.2118/17369-MS](https://doi.org/10.2118/17369-MS).
- [25] K. Bao, K.-A. Lie, O. Møyner, and M. Liu. Fully-implicit simulation of polymer flooding with mrst. In *ECMOR XV – 15th European Conference on the Mathematics of Oil Recovery*. EAGE, 2016. doi:[10.3997/2214-4609.201601880](https://doi.org/10.3997/2214-4609.201601880).
- [26] K. Bao, K.-A. Lie, O. Møyner, and M. Liu. Fully implicit simulation of polymer flooding with mrst. *Comput. Geosci.*, 2017. doi:[10.1007/s10596-017-9624-5](https://doi.org/10.1007/s10596-017-9624-5).
- [27] J. Barker and S. Thibeau. A critical review of the use of pseudorelative permeabilities for upscaling. *SPE Reservoir Engineering*, 12(2):138–143, 1997. doi:[10.2118/35491-PA](https://doi.org/10.2118/35491-PA).
- [28] R. P. Batycky, M. R. Thieles, R. O. Baker, and S. H. Chugh. Revisiting reservoir flood-surveillance methods using streamlines. *SPE Res. Eval. Eng.*, 11(2):387–394, 2008. doi:[10.2118/95402-PA](https://doi.org/10.2118/95402-PA).
- [29] D. Baxendale. *Open Porous Media: Flow documentation manual*, 2017-10 rev-0 edition, 2017. URL <http://opm-project.org>.
- [30] J. Bear. *Dynamics of Fluids in Porous Media*. Dover, 1988. ISBN 0-486-45355-3.
- [31] J. Bear. *Hydraulics of Groundwater*. Dover, 2007. ISBN 0-486-65675-6.
- [32] J. Bear and Y. Bachmat. *Introduction to Modeling of Transport Phenomena in Porous Media*. Theory and Applications of Transport in Porous Media. Springer, 1990. ISBN 9780792305576.
- [33] S. H. Begg, R. R. Carter, and P. Dranfield. Assigning effective values to simulator gridblock parameters for heterogeneous reservoirs. *SPE Res. Eng.*, 4(4):455–463, 1989.
- [34] L. Beirão da Veiga, F. Brezzi, A. Cangiani, G. Manzini, L. D. Marini, and A. Russo. Basic principles of virtual element methods. *Math. Mod. Meth. Appl. Sci.*, 23(01):199–214, 2013. doi:[10.1142/S0218202512500492](https://doi.org/10.1142/S0218202512500492).
- [35] L. Beirão da Veiga, F. Brezzi, L. D. Marini, and A. Russo. The hitchhiker’s guide to the virtual element method. *Math. Mod. Meth. Appl. Sci.*, 24(08):1541–1573, 2014. doi:[10.1142/S021820251440003X](https://doi.org/10.1142/S021820251440003X).
- [36] 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](https://doi.org/10.1007/978-3-319-02663-3).
- [37] A. Benesoussan, J.-L. Lions, and G. Papanicolaou. *Asymptotic Analysis for Periodic Structures*. Elsevier Science Publishers, Amsterdam, 1978.
- [38] R. L. Berge. Unstructured PEI grids adapting to geological features in subsurface reservoirs. Master’s thesis, Norwegian University of Science and Technology, 2016.

- [39] R. L. Berge. Unstructured PEBI-grids adapting to geological features in subsurface reservoirs. Master's thesis, Norwegian University of Science and Technology, 2016. URL <http://hdl.handle.net/11250/2411565>.
- [40] 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](https://doi.org/10.1109/SCAM.2002.1134106).
- [41] D. Braess. *Finite elements: Theory fast solvers and applications in solid mechanics*. Cambridge University Press, Cambridge, 1997.
- [42] L. Branets, S. S. Ghai, S. L. Lyons, and X.-H. Wu. Efficient and accurate reservoir modeling using adaptive gridding with global scale up. In *SPE Reservoir Simulation Symposium, 2–4 February, The Woodlands, Texas, 2009*. doi:[10.2118/118946-MS](https://doi.org/10.2118/118946-MS).
- [43] Y. Brenier and J. Jaffré. Upstream differencing for multiphase flow in reservoir simulation. *SIAM J. Numer. Anal.*, 28(3):685–696, 1991. doi:[10.1137/0728036](https://doi.org/10.1137/0728036).
- [44] 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.
- [45] M. Brewer, D. Camilleri, S. Ward, and T. Wong. Generation of hybrid grids for simulation of complex, unstructured reservoirs by a simulator with mpfa. In *SPE Reservoir Simulation Symposium, 23–25 February, Houston, Texas, USA*, 2015. doi:[10.2118/173191-MS](https://doi.org/10.2118/173191-MS).
- [46] 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.
- [47] F. Brezzi, K. Lipnikov, and V. Simoncini. A family of mimetic finite difference methods on polygonal and polyhedral meshes. *Math. Models Methods Appl. Sci.*, 15:1533–1553, 2005. doi:[10.1142/S0218202505000832](https://doi.org/10.1142/S0218202505000832).
- [48] R. H. Brooks and A. T. Corey. Properties of porous media affecting fluid flow. *J. Irrigation Drainage Div.*, 92(2):61–90, 1966.
- [49] E. Buckingham. *Studies on the movement of soil moisture*. Number 38. United States. Bureau of Soils, 1907.
- [50] H. Cao. *Development of techniques for general purpose simulators*. PhD thesis, Stanford University, 2002.
- [51] Cayuga Research. Admat. URL <http://www.cayugaresearch.com/admat.html>. [Online; accessed 15-04-2014].
- [52] G. Chavent and J. Jaffré. *Mathematical models and finite elements for reservoir simulation*. North Holland, 1982.
- [53] G. Chavent and J. Jaffré. *Mathematical models and finite elements for reservoir simulation: single phase, multiphase and multicomponent flows*

- through porous media.* Studies in Mathematics and its Applications. Elsevier, 1986.
- [54] Y. Chen and L. J. Durlofsky. Adaptive local-global upscaling for general flow scenarios in heterogeneous formations. *Transport Porous Media*, 62: 157–182, 2006.
- [55] Y. Chen, L. J. Durlofsky, M. Gerritsen, and X. H. Wen. A coupled local-global upscaling approach for simulating flow in highly heterogeneous formations. *Adv. Water Resour.*, 26:1041–1060, 2003.
- [56] Z. Chen. Formulations and numerical methods of the black oil model in porous media. *SIAM J. Numer. Anal.*, 38(2):489–514, 2000. doi:[10.1137/S0036142999304263](https://doi.org/10.1137/S0036142999304263).
- [57] Z. Chen. *Reservoir Simulation: Mathematical Techniques in Oil Recovery*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2007.
- [58] Z. Chen and R. E. Ewing. Comparison of various formulations of three-phase flow in porous media. *J. Comput. Phys.*, 132(2):362–373, 1997. doi:[10.1006/jcph.1996.5641](https://doi.org/10.1006/jcph.1996.5641).
- [59] Z. Chen and T. Y. Hou. A mixed multiscale finite element method for elliptic problems with oscillating coefficients. *Math. Comp.*, 72:541–576, 2003. doi:[10.1090/S0025-5718-02-01441-2](https://doi.org/10.1090/S0025-5718-02-01441-2).
- [60] 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](https://doi.org/10.1137/1.9780898718942).
- [61] M. A. Christie. Upscaling for reservoir simulation. *J. Pet. Tech.*, 48 (11):1004–1010, 1996. doi:[10.2118/37324-MS](https://doi.org/10.2118/37324-MS).
- [62] 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](https://doi.org/10.2118/72469-PA). Url: <http://www.spe.org/csp/>.
- [63] K. H. Coats. A note on IMPES and some IMPES-based simulation models. *SPE J.*, 05(03):245–251, 2000. doi:[10.2118/65092-PA](https://doi.org/10.2118/65092-PA).
- [64] 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.
- [65] R. Courant, K. Friedrichs, and H. Lewy. Über die partiellen Differenzengleichungen der mathematischen Physik. *Math. Ann.*, 100(1):32–74, 1928.
- [66] C. M. Dafermos. *Hyperbolic Conservation Laws in Continuum Physics*, volume 325 of *Grundlehren der mathematischen Wissenschaften*. Springer Berlin Heidelberg, 2010. doi:[10.1007/978-3-642-04048-1](https://doi.org/10.1007/978-3-642-04048-1).
- [67] H. P. G. Darcy. *Les Fontaines Publiques de la Ville de Dijon*. Dalmont, Paris, 1856.

- [68] A. Datta-Gupta and M. J. King. A semianalytic approach to tracer flow modeling in heterogeneous permeable media. *Adv. Water Resour.*, 18:9–24, 1995.
- [69] A. Datta-Gupta and M. J. King. *Streamline Simulation: Theory and Practice*, volume 11 of *SPE Textbook Series*. Society of Petroleum Engineers, 2007.
- [70] D. DeBaun, T. Byer, P. Childs, J. Chen, F. Saaf, M. Wells, J. Liu, H. Cao, L. Pianello, V. Tilakraj, P. Crumpton, D. Walsh, H. Yardumian, R. Zorzynski, K.-T. Lim, M. Schrader, V. Zapata, J. Nolen, and H. A. Tchelepi. An extensible architecture for next generation scalable parallel reservoir simulation. In *SPE Reservoir Simulation Symposium, 31 January–2 February, The Woodlands, Texas, USA*, 2005. doi:[10.2118/93274-MS](https://doi.org/10.2118/93274-MS).
- [71] C. V. Deutsch and A. G. Journel. *GSLIB: Geostatistical software library and user's guide*. Oxford University Press, New York, 2nd edition, 1998.
- [72] X. Y. Ding and L. S. K. Fung. An unstructured gridding method for simulating faulted reservoirs populated with complex wells. In *SPE Reservoir Simulation Symposium, 23–25 February, Houston, Texas, USA*, 2015. doi:[10.2118/173243-MS](https://doi.org/10.2118/173243-MS).
- [73] J. Douglas Jr, D. W. Peaceman, and H. H. Rachford Jr. A method for calculating multi-dimensional immiscible displacement. *Trans. AIME*, 216:297–308, 1959.
- [74] A. C. Duarte. *Contributions to production optimization of oil reservoirs*. PhD thesis, Norwegian University of Science and Technology, 2016. URL <http://hdl.handle.net/11250/2383090>.
- [75] I. S. Duff. Ma57—a code for the solution of sparse symmetric definite and indefinite systems. *ACM Trans. Math. Softw.*, 30(2):118–144, June 2004. doi:[10.1145/992200.992202](https://doi.org/10.1145/992200.992202).
- [76] I. S. Duff and J. K. Reid. The multifrontal solution of indefinite sparse symmetric linear. *ACM Trans. Math. Software*, 9(3):302–325, 1983. doi:[10.1145/356044.356047](https://doi.org/10.1145/356044.356047).
- [77] L. J. Durlofsky. Numerical calculations of equivalent gridblock permeability tensors for heterogeneous porous media. *Water Resour. Res.*, 27(5):699–708, 1991.
- [78] L. J. Durlofsky. Upscaling of geocellular models for reservoir flow simulation: A review of recent progress, 2003. Presented at 7th International Forum on Reservoir Simulation Bühl/Baden-Baden, Germany, June 23–27, 2003.
- [79] L. J. Durlofsky. Upscaling and gridding of fine scale geological models for flow simulation, 2005. Presented at 8th International Forum on Reservoir Simulation Iles Borromées, Stresa, Italy, June 20–24, 2005.
- [80] 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.

- [81] 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.
- [82] 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>.
- [83] 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](https://doi.org/10.1007/s10596-009-9153-y).
- [84] T. Ertekin, J. H. Abou-Kassem, and G. R. King. *Basic applied reservoir simulation*, volume 7 of *SPE Textbook Series*. Society of Petroleum Engineers Richardson, TX, 2001.
- [85] 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](https://doi.org/10.1023/A:1011543412675).
- [86] J. R. Fanchi. *Principles of applied reservoir simulation*. Gulf Professional Publishing, 2005.
- [87] C. L. Farmer. Upscaling: a review. *Int. J. Numer. Meth. Fluids*, 40(1–2):63–78, 2002. doi:[10.1002/fld.267](https://doi.org/10.1002/fld.267).
- [88] 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].
- [89] F. J. T. Floris, M. D. Bush, M. Cuypers, F. Roggero, and A. R. Syversveen. Methods for quantifying the uncertainty of production forecasts: a comparative study. *Petroleum Geoscience*, 7(S):S87–S96, 2001.
- [90] S. A. Forth. An efficient overloaded implementation of forward mode automatic differentiation in MATLAB. *ACM Trans. Math. Software*, 32(2):195–222, 2006.
- [91] L. S. K. Fung, X. Y. Ding, , and A. H. Dogru. Unconstrained voronoi grids for densely spaced complex wells in full-field reservoir simulation. *SPE J.*, 19(5):803–815, 2014. doi:[10.2118/163648-PA](https://doi.org/10.2118/163648-PA).
- [92] A. L. Gain, C. Talischi, and G. H. Paulino. On the virtual element method for three-dimensional linear elasticity problems on arbitrary polyhedral meshes. *Comput. Meth. App. Mech. Engng.*, 282:132–160, 2014.
- [93] M. Gerritsen and J. V. Lambers. Integration of local-global upscaling and grid adaptivity for simulation of subsurface flow in heterogeneous formations. *Comput. Geosci.*, 12(2):193–208, 2008. doi:[10.1007/s10596-007-9078-2](https://doi.org/10.1007/s10596-007-9078-2).

- [94] S. K. Godunov. A difference method for numerical calculation of discontinuous solutions of the equations of hydrodynamics. *Mat. Sb. (N.S.)*, 47 (89):271–306, 1959.
- [95] S. Gries, K. Stüben, G. L. Brown, D. Chen, and D. A. Collins. Preconditioning for efficiently applying algebraic multigrid in fully implicit reservoir simulations. *SPE J.*, 19(04):726–736, 2014. doi:[10.2118/163608-PA](https://doi.org/10.2118/163608-PA).
- [96] D. Guéritot, J. L. Rudkiewicz, C. Ravenne, D. Renard, and A. Galli. An integrated model for computer aided reservoir description: From outcrop study to fluid flow simulations. *Oil & Gas Science and Technology*, 45 (1):71–77, 1990.
- [97] H. Hægland, H. K. Dahle, K.-A. Lie, and G. T. Eigestad. Adaptive streamline tracing for streamline simulation on irregular grids. In P. Bining, 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/>.
- [98] H. Hajibeygi and H. A. Tchelepi. Compositional multiscale finite-volume formulation. *SPE J.*, 19(2):316–326, 2014. doi:[10.2118/163664-PA](https://doi.org/10.2118/163664-PA).
- [99] H. B. Hales. A method for creating 2-d orthogonal grids which conform to irregular shapes. *SPE J.*, 1(2):115–124, 1996. doi:[10.2118/35273-PA](https://doi.org/10.2118/35273-PA).
- [100] A. Harten, P. D. Lax, and B. v. Leer. On upstream differencing and Godunov-type schemes for hyperbolic conservation laws. *SIAM review*, 25(1):35–61, 1983. doi:[10.1137/1025002](https://doi.org/10.1137/1025002).
- [101] V. L. Hauge. *Multiscale Methods and Flow-based Gridding for Flow and Transport in Porous Media*. PhD thesis, Norwegian University of Science and Technology, 2010. URL <http://ntnu.diva-portal.org/smash/get/diva2:400507/FULLTEXT02>.
- [102] V. L. Hauge, K.-A. Lie, and J. R. Natvig. Flow-based coarsening for multiscale simulation of transport in porous media. *Comput. Geosci.*, 16(2):391–408, 2012. doi:[10.1007/s10596-011-9230-x](https://doi.org/10.1007/s10596-011-9230-x).
- [103] Z. He, H. Parikh, A. Datta-Gupta, J. Perez, and T. Pham. Identifying reservoir compartmentalization and flow barriers from primary production using streamline diffusive time of flight. *SPE J.*, 7(3):238–247, June 2004. doi:[10.2118/88802-PA](https://doi.org/10.2118/88802-PA).
- [104] R. Helmig. *Multiphase flow and transport processes in the subsurface: a contribution to the modeling of hydroystems*. Environmental engineering. Springer, 1997. ISBN 9783540627036.
- [105] S. T. Hilden. *Upscaling of water-flooding scenarios and modeling of polymer flow*. PhD thesis, Norwegian University of Science and Technology, 2016. URL <http://hdl.handle.net/11250/2388331>.
- [106] S. T. Hilden, K.-A. Lie, and X. Raynaud. Steady state upscaling of polymer flooding. In *ECMOR XIV – 14th European Conference on the Mathematics of Oil Recovery, Catania, Sicily, Italy, 8-11 September 2014*. EAGE, 2014. doi:[10.3997/2214-4609.20141802](https://doi.org/10.3997/2214-4609.20141802).

- [107] H. Holden and N. Risebro. *Front Tracking for Hyperbolic Conservation Laws*, volume 152 of *Applied Mathematical Sciences*. Springer, New York, 2002.
- [108] H. Holden and N. H. Risebro. *Front Tracking for Hyperbolic Conservation Laws*, volume 152 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 2002. ISBN 3-540-43289-2.
- [109] L. Holden and B. F. Nielsen. Global upscaling of permeability in heterogeneous reservoirs; the output least squares (ols) method. *Trans. Porous Media*, 40(2):115–143, 2000.
- [110] U. Hornung. *Homogenization and porous media*. Springer-Verlag, New York, 1997.
- [111] H. Hoteit and A. Firoozabadi. Numerical modeling of two-phase flow in heterogeneous permeable media with different capillarity pressures. *Adv. Water Resour.*, 31(1):56–73, 2008. doi:[10.1016/j.advwatres.2007.06.006](https://doi.org/10.1016/j.advwatres.2007.06.006).
- [112] M. K. Hubbert. Darcy's law and the field equations of the flow of underground fluids. *Petrol. Trans., AIME*, 207:22–239, 1956.
- [113] E. Idrobo, M. Choudhary, and A. Datta-Gupta. Swept volume calculations and ranking of geostatistical reservoir models using streamline simulation. In *SPE/AAPG Western Regional Meeting*, Long Beach, California, USA, 19–23 June 2000. SPE 62557.
- [114] M. R. Islam, S. H. Mousavizadegan, S. Mustafiz, and J. H. Abou-Kassem. *Advanced Petroleum Reservoir Simulations*. John Wiley & Sons, Inc., 2010. ISBN 9780470650684. doi:[10.1002/9780470650684](https://doi.org/10.1002/9780470650684).
- [115] O. Izgec, M. Sayarpour, and G. M. Shook. Maximizing volumetric sweep efficiency in waterfloods with hydrocarbon $f\phi$ curves. *Journal of Petroleum Science and Engineering*, 78(1):54–64, 2011. doi:[10.1016/j.petrol.2011.05.003](https://doi.org/10.1016/j.petrol.2011.05.003).
- [116] J. D. Jansen. *Nodal analysis of oil and gas wells system modeling and numerical implementation*. SPE Textbook Series. Society of Petroleum Engineers, Richardson, 2017.
- [117] P. Jenny, C. Wolfsteiner, S. H. Lee, and L. J. Durlofsky. Modeling flow in geometrically complex reservoirs using hexahedral multiblock grids. *SPE J.*, 7(2), 2002. doi:[10.2118/78673-PA](https://doi.org/10.2118/78673-PA).
- [118] P. Jenny, S. H. Lee, and H. A. Tchelepi. Multi-scale finite-volume method for elliptic problems in subsurface flow simulation. *J. Comput. Phys.*, 187:47–67, 2003. doi:[10.1016/S0021-9991\(03\)00075-5](https://doi.org/10.1016/S0021-9991(03)00075-5).
- [119] P. Jenny, H. A. Tchelepi, and S. H. Lee. Unconditionally convergent nonlinear solver for hyperbolic conservation laws with s-shaped flux functions. *J. Comput. Phys.*, 228(20):7497–7512, 2009. doi:[10.1016/j.jcp.2009.06.032](https://doi.org/10.1016/j.jcp.2009.06.032).
- [120] V. V. Jikov, S. M. Kozlov, and O. A. Oleinik. *Homogenization of differential operators and integral functionals*. Springer-Verlag, New York, 1994.

- [121] E. Jimenez, K. Sabir, A. Datta-Gupta, and M. J. King. Spatial error and convergence in streamline simulation. *SPE J.*, 10(3):221–232, June 2007.
- [122] A. Journal, C. Deutsch, and A. Desbarats. Power averaging for block effective permeability. In *SPE California Regional Meeting, 2-4 April, Oakland, California*, SPE 15128, 1986.
- [123] R. Juanes and T. W. Patzek. Three-phase displacement theory: an improved description of relative permeabilities. *SPE J.*, 9(3):302–313, 2004. doi:[10.2118/88973-PA](https://doi.org/10.2118/88973-PA).
- [124] R. Juanes and T. W. Patzek. Relative permeabilities for strictly hyperbolic models of three-phase flow in porous media. *Transp. Porous Media*, 57(2):125–152, 2004. doi:[10.1023/B:TIPM.0000038251.10002.5e](https://doi.org/10.1023/B:TIPM.0000038251.10002.5e).
- [125] M. Karimi-Fard and L. J. Durlofsky. Accurate resolution of near-well effects in upscaled models using flow-based unstructured local grid refinement. *SPE J.*, 17(4):1084–1095, 2012. doi:[10.2118/141675-PA](https://doi.org/10.2118/141675-PA).
- [126] K. Karlsen, K.-A. Lie, and N. Risebro. A front tracking method for conservation laws with boundary conditions. In M. Fey and R. Jeltsch, editors, *Hyperbolic Problems: Theory, Numerics, Applications*, volume 129 of *International Series of Numerical Mathematics*, pages 493–502. Birkhäuser Basel, 1999. doi:[10.1007/978-3-0348-8720-5_53](https://doi.org/10.1007/978-3-0348-8720-5_53).
- [127] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comp.*, 20(1):359–392, 1998. doi:[10.1137/S1064827595287997](https://doi.org/10.1137/S1064827595287997).
- [128] E. Keilegavlen and J. M. Nordbotten. Finite volume methods for elasticity with weak symmetry. *arXiv preprint arXiv:1512.01042*, 2015.
- [129] E. Keilegavlen, J. E. Kozdon, and B. T. Mallison. Multidimensional upstream weighting for multiphase transport on general grids. *Comput. Geosci.*, 16:1021–1042, 2012. doi:[10.1007/s10596-012-9301-7](https://doi.org/10.1007/s10596-012-9301-7).
- [130] D. Kenyon. Third SPE comparative solution project: gas cycling of retrograde condensate reservoirs. *J. Petrol. Tech.*, 39(08):981–997, 1987. doi:[10.2118/12278-PA](https://doi.org/10.2118/12278-PA).
- [131] J. E. Killough. Ninth SPE comparative solution project: A reexamination of black-oil simulation. In *SPE Reservoir Simulation Symposium, 12-15 February 1995, San Antonio, Texas*, 1995. doi:[10.2118/29110-MS](https://doi.org/10.2118/29110-MS). SPE 29110-MS.
- [132] M. King, D. MacDonald, S. Todd, and H. Leung. Application of novel upscaling approaches to the Magnus and Andrew reservoirs. In *European Petroleum Conference, 20-22 October, The Hague, Netherlands*, SPE 50643, 1998.
- [133] M. J. King and A. Datta-Gupta. Streamline simulation: A current perspective. *In Situ*, 22(1):91–140, 1998.
- [134] M. J. King and M. Mansfield. Flow simulation of geologic models. *SPE Res. Eval. Eng.*, 2(4):351–367, 1999. doi:[10.2118/57469-PA](https://doi.org/10.2118/57469-PA).

- [135] V. Kippe, J. E. Aarnes, and K.-A. Lie. A comparison of multiscale methods for elliptic problems in porous media flow. *Comput. Geosci.*, 12(3):377–398, 2008. ISSN 1420-0597. doi:[10.1007/s10596-007-9074-6](https://doi.org/10.1007/s10596-007-9074-6).
- [136] 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.
- [137] R. A. Klausen, A. F. Rasmussen, and A. Stephansen. Velocity interpolation and streamline tracing on irregular geometries. *Comput. Geosci.*, 16:261–276, 2012. doi:[10.1007/s10596-011-9256-0](https://doi.org/10.1007/s10596-011-9256-0).
- [138] Ø. S. Klemetsdal. The virtual element method as a common framework for finite element and finite difference methods – numerical and theoretical analysis. Master’s thesis, Norwegian university of science and technology, 2016.
- [139] S. Krogstad, K.-A. Lie, O. Møyner, H. M. Nilsen, X. Raynaud, and B. Skaflestad. MRST-AD – an open-source framework for rapid prototyping and evaluation of reservoir simulation problems. In *SPE Reservoir Simulation Symposium, 23–25 February, Houston, Texas*, 2015. doi:[10.2118/173317-MS](https://doi.org/10.2118/173317-MS).
- [140] S. Krogstad, X. Raynaud, and H. M. Nilsen. Reservoir management optimization using well-specific upscaling and control switching. *Comput. Geosci.*, 2015. doi:[10.1007/s10596-015-9497-4](https://doi.org/10.1007/s10596-015-9497-4).
- [141] S. Krogstad, K.-A. Lie, H. M. Nilsen, C. F. Berg, and V. Kippe. Flow diagnostics for optimal polymer injection strategies. In *ECMOR XV – 15th European conference on the mathematics of oil recovery, Amsterdam, Netherlands*. EAGE, 2016. doi:[10.3997/2214-4609.201601874](https://doi.org/10.3997/2214-4609.201601874).
- [142] S. N. Kružkov. First order quasilinear equations in several independent variables. *Mathematics of the USSR-Sbornik*, 10(2):217, 1970.
- [143] A. Kurganov, S. Noelle, and G. Petrova. Semidiscrete central-upwind schemes for hyperbolic conservation laws and hamilton-jacobi equations. *SIAM J. Sci. Comp.*, 23(3):707–740, 2001. doi:[10.1137/S1064827500373413](https://doi.org/10.1137/S1064827500373413).
- [144] F. Kwok and H. Tchelepi. Potential-based reduced Newton algorithm for nonlinear multiphase flow in porous media. *J. Comput. Phys.*, 227(1):706–727, 2007. doi:[10.1016/j.jcp.2007.08.012](https://doi.org/10.1016/j.jcp.2007.08.012).
- [145] L. W. Lake. *Enhanced Oil Recovery*. Prentice-Hall, 1989.
- [146] P. Lax and B. Wendroff. Systems of conservation laws. *Comm. Pure Appl. Math.*, 13(2):217–237, 1960.
- [147] S. H. Lee, P. Jenny, and H. A. Tchelepi. A finite-volume method with hexahedral multiblock grids for modeling flow in porous media. *Comput. Geosci.*, 6(3-4):353–379, 2002. ISSN 1420-0597. Locally conservative numerical methods for flow in porous media.
- [148] O. Leeuwenburgh and R. Arts. Distance parameterization for efficient seismic history matching with the ensemble kalman filter. In *ECMOR XIII - 13th European Conference on the Mathematics of Oil Recovery*. EAGE, 2012. doi:[10.3997/2214-4609.20143176](https://doi.org/10.3997/2214-4609.20143176).

- [149] O. Leeuwenburgh, E. Peters, and F. Wilschut. Towards an integrated workflow for structural reservoir model updating and history matching. In *SPE EUROPEC/EAGE Annual Conference and Exhibition, 23-26 May, Vienna, Austria*, 2011. doi:[10.2118/143576-MS](https://doi.org/10.2118/143576-MS).
- [150] R. J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge University Press, Cambridge, 2002.
- [151] M. C. Leverett. Capillary behavior in porous solids. *Trans. AIME*, 142: 159–172, 1941. doi:[10.2118/941152-G](https://doi.org/10.2118/941152-G).
- [152] X. Li and D. Zhang. A backward automatic differentiation framework for reservoir simulation. *Comput. Geosci.*, pages 1–14, 2014. doi:[10.1007/s10596-014-9441-z](https://doi.org/10.1007/s10596-014-9441-z).
- [153] K. 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. ISSN 1420-0597. doi:[10.1007/s10596-011-9244-4](https://doi.org/10.1007/s10596-011-9244-4). URL <http://dx.doi.org/10.1007/s10596-011-9244-4>.
- [154] K.-A. Lie. *Jolt 1: Introduction to MRST*. SINTEF ICT / ICME, Stanford University, Dec. 2015. URL <http://www.sintef.no/projectweb/mrst/jolts/>. Jolts – Just-in-time online learning tools.
- [155] K.-A. Lie. *Jolt 2: Grids and petrophysical data*. SINTEF ICT / ICME, Stanford University, Dec. 2015. URL <http://www.sintef.no/projectweb/mrst/jolts/>. Jolts – Just-in-time online learning tools.
- [156] K.-A. Lie, S. Krogstad, I. S. Ligaarden, J. R. Natvig, H. M. Nilsen, and B. Skaflestad. Discretisation on complex grids – open source MATLAB implementation. In *Proceedings of ECMOR XII–12th European Conference on the Mathematics of Oil Recovery*, Oxford, UK, 6–9 September 2010. EAGE.
- [157] K.-A. Lie, J. R. Natvig, and H. M. Nilsen. Discussion of dynamics and operator splitting techniques for two-phase flow with gravity. *Int. J. Numer. Anal. Mod. (Special issue in memory of Magne Espedal)*, 9(3): 684–700, 2012.
- [158] K.-A. Lie, J. R. Natvig, S. Krogstad, Y. Yang, and X.-H. Wu. Grid adaptation for the Dirichlet–Neumann representation method and the multiscale mixed finite-element method. *Comput. Geosci.*, 18(3):357–372, 2014. doi:[10.1007/s10596-013-9397-4](https://doi.org/10.1007/s10596-013-9397-4).
- [159] K.-A. Lie, H. M. Nilsen, O. Andersen, and O. Møyner. A simulation workflow for large-scale CO₂ storage in the Norwegian North Sea. In *ECMOR XIV – 14th European Conference on the Mathematics of Oil Recovery, Catania, Sicily, Italy, 8-11 September 2014*. EAGE, 2014. doi:[10.3997/2214-4609.20141877](https://doi.org/10.3997/2214-4609.20141877).
- [160] K.-A. Lie, H. M. Nilsen, O. Andersen, and O. Møyner. A simulation workflow for large-scale CO₂ storage in the Norwegian North Sea. *Comput. Geosci.*, :1–16, 2015. doi:[10.1007/s10596-015-9487-6](https://doi.org/10.1007/s10596-015-9487-6).

- [161] K.-A. Lie, O. Møyner, J. R. Natvig, A. Kozlova, K. Bratvedt, S. Watanabe, and Z. Li. Successful application of multiscale methods in a real reservoir simulator environment. In *ECMOR XV – 15th European Conference on the Mathematics of Oil Recovery, Amsterdam, Netherlands, 29 Aug–1 Sept, 2016*. doi:[10.3997/2214-4609.201601893](https://doi.org/10.3997/2214-4609.201601893).
- [162] K.-A. Lie, K. Kedia, B. Skaflestad, X. Wang, Y. Yang, X.-H. Wu, and N. Hoda. a general non-uniform coarsening and upscaling framework for reduced-order modeling. In *SPE Reservoir Simulation Conference, Montgomery, Texas, USA, 20-22 February 2017*, 2017. doi:[10.2118/182681-MS](https://doi.org/10.2118/182681-MS).
- [163] K.-A. Lie, O. Møyner, and J. R. Natvig. A feature-enriched multiscale method for simulating complex geomodels. In *SPE Reservoir Simulation Conference, Montgomery, Texas, USA, 20–22 February 2017*, 2017. doi:[10.2118/182701-MS](https://doi.org/10.2118/182701-MS).
- [164] I. S. Ligaarden. Well models for mimetic finite difference methods and improved representation of wells in multiscale methods. Master's thesis, University of Oslo, 2008. URL <http://www.duo.uio.no/sok/work.html?WORKID=77236>.
- [165] K. Lipnikov, M. Shashkov, and I. Yotov. Local flux mimetic finite difference methods. *Numer. Math.*, 112(1):115–152, 2009. ISSN 0029-599X. doi:[10.1007/s00211-008-0203-5](https://doi.org/10.1007/s00211-008-0203-5). URL <http://dx.doi.org/10.1007/s00211-008-0203-5>.
- [166] I. Lunati and S. H. Lee. An operator formulation of the multiscale finite-volume method with correction function. *Multiscale Model. Simul.*, 8(1):96–109, 2009. doi:[10.1137/080742117](https://doi.org/10.1137/080742117).
- [167] B. Mallison, C. Sword, T. Viard, W. Milliken, and A. Cheng. Unstructured cut-cell grids for modeling complex reservoirs. *SPE J.*, 2014. doi:[10.2118/163642-PA](https://doi.org/10.2118/163642-PA).
- [168] 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.
- [169] S. F. Matringe and M. G. Gerritsen. On accurate tracing of streamlines. In *SPE Annual Technical Conference and Exhibition*, Houston, Texas, USA, 26-29 September 2004. SPE 89920.
- [170] S. F. Matringe, R. Juanes, and H. A. Tchelepi. Streamline tracing on general triangular or quadrilateral grids. *SPE J.*, 12(2):217–233, June 2007.
- [171] C. C. Mattax and R. L. Dalton, editors. *Reservoir Simulation*, volume 13 of *SPE Monograph Series*. Society of Petroleum Engineers, 1990. ISBN 978-1-55563-028-7.
- [172] W. D. McCain, Jr. *The properties of petroleum fluids*. PennWell Books, Tulsa, Oklahoma, 2nd edition, 1990.
- [173] 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].
- [174] R. Merland, G. Caumon, B. Lvy, and P. Collon-Drouaillet. Voronoi grids conforming to 3d structural features. *Comput. Geosci.*, 18(3-4):373–383, 2014. doi:[10.1007/s10596-014-9408-0](https://doi.org/10.1007/s10596-014-9408-0).
- [175] O. Møyner. Multiscale finite-volume methods on unstructured grids. Master’s thesis, Norwegian University of Science and Technology, Trondheim, 2012. URL <http://daim.idi.ntnu.no/masteroppgave?id=7377>.
- [176] O. Møyner. *Next generation multiscale methods for reservoir simulation*. PhD thesis, Norwegian University of Science and Technology, 2016. URL <http://hdl.handle.net/11250/2431831>.
- [177] O. Møyner and K.-A. Lie. The multiscale finite-volume method on stratigraphic grids. *SPE J.*, 19(5):816–831, 2014. doi:[10.2118/163649-PA](https://doi.org/10.2118/163649-PA).
- [178] O. Møyner and K.-A. Lie. A multiscale method based on restriction-smoothed basis functions suitable for general grids in high contrast media. In *SPE Reservoir Simulation Symposium held in Houston, Texas, USA, 23–25 February 2015*, 2015. doi:[10.2118/173256-MS](https://doi.org/10.2118/173256-MS). SPE 173265-MS.
- [179] O. Møyner and K.-A. Lie. A multiscale restriction-smoothed basis method for high contrast porous media represented on unstructured grids. *J. Comput. Phys.*, 304:46–71, 2016. doi:[10.1016/j.jcp.2015.10.010](https://doi.org/10.1016/j.jcp.2015.10.010).
- [180] O. Møyner and K.-A. Lie. A multiscale restriction-smoothed basis method for compressible black-oil models. *SPE J.*, 2016. in press.
- [181] O. Møyner and K.-A. Lie. A multiscale restriction-smoothed basis method for high contrast porous media represented on unstructured grids. *J. Comput. Phys.*, 2016. doi:[10.1016/j.jcp.2015.10.010](https://doi.org/10.1016/j.jcp.2015.10.010).
- [182] O. Møyner and K.-A. Lie. A multiscale restriction-smoothed basis method for compressible black-oil models. *SPE J.*, 21(06), 2016. doi:[10.2118/173265-PA](https://doi.org/10.2118/173265-PA).
- [183] O. Møyner and H. A. Tchelepi. A feature-enriched multiscale method for simulating complex geomodels. In *SPE Reservoir Simulation Conference, Montgomery, Texas, USA, 20–22 February 2017*, 2017. doi:[10.2118/182679-MS](https://doi.org/10.2118/182679-MS).
- [184] O. Møyner, S. Krogstad, and K.-A. Lie. The application of flow diagnostics for reservoir management. *SPE J.*, 20(2):306–323, 2014. doi:[10.2118/171557-PA](https://doi.org/10.2118/171557-PA).
- [185] M. Muskat and R. D. Wyckoff. *The flow of homogeneous fluids through porous media*, volume 12. McGraw-Hill New York, 1937.
- [186] 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](https://doi.org/10.1016/j.jcp.2008.08.024).

- [187] 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](https://doi.org/10.1016/j.advwatres.2007.05.015).
- [188] J. R. Natvig, B. Skaflestad, F. Bratvedt, K. Bratvedt, K.-A. Lie, V. Laptev, and S. K. Khataniar. Multiscale mimetic solvers for efficient streamline simulation of fractured reservoirs. *SPE J.*, 16(4):880–888, 2011. doi:[10.2018/119132-PA](https://doi.org/10.2018/119132-PA).
- [189] J. R. Natvig, K.-A. Lie, S. Krogstad, Y. Yang, and X.-H. Wu. Grid adaptation for upscaling and multiscale methods. In *Proceedings of ECMOR XIII–13th European Conference on the Mathematics of Oil Recovery*, Biarritz, France, 10–13 September 2012. EAGE.
- [190] R. Neidinger. Introduction to automatic differentiation and MATLAB object-oriented programming. *SIAM Review*, 52(3):545–563, 2010. doi:[10.1137/080743627](https://doi.org/10.1137/080743627).
- [191] H. Nessyahu and E. Tadmor. Nonoscillatory central differencing for hyperbolic conservation laws. *J. Comput. Phys.*, 87(2):408–463, 1990.
- [192] B. F. Nielsen and A. Tveito. An upscaling method for one-phase flow in heterogeneous reservoirs; a Weighted Output Least Squares (WOLS) approach. *Comput. Geosci.*, 2:92–123, 1998.
- [193] H. M. Nilsen, P. A. Herrera, M. Ashraf, I. Ligaarden, M. Iding, C. Hermannrud, K.-A. Lie, J. M. Nordbotten, H. K. Dahle, and E. Keilegavlen. Field-case simulation of CO₂-plume migration using vertical-equilibrium models. *Energy Procedia*, 4(0):3801–3808, 2011. doi:[10.1016/j.egypro.2011.02.315](https://doi.org/10.1016/j.egypro.2011.02.315).
- [194] H. M. Nilsen, K.-A. Lie, and J. R. Natvig. Accurate modelling of faults by multipoint, mimetic, and mixed methods. *SPE J.*, 17(2):568–579, 2012. doi:[10.2118/149690-PA](https://doi.org/10.2118/149690-PA).
- [195] H. M. Nilsen, K.-A. Lie, and O. Andersen. Robust simulation of sharp-interface models for fast estimation of CO₂ trapping capacity. *Computational Geosciences*, pages 1–21, 2015. ISSN 1420-0597. doi:[10.1007/s10596-015-9549-9](https://doi.org/10.1007/s10596-015-9549-9). URL <http://dx.doi.org/10.1007/s10596-015-9549-9>.
- [196] H. M. Nilsen, K.-A. Lie, and O. Andersen. Fully implicit simulation of vertical-equilibrium models with hysteresis and capillary fringe. *Comput. Geosci.*, : , 2015. ISSN 1420-0597. doi:[10.1007/s10596-015-9547-y](https://doi.org/10.1007/s10596-015-9547-y). URL <http://dx.doi.org/10.1007/s10596-015-9547-y>.
- [197] H. M. Nilsen, K.-A. Lie, and O. Andersen. Analysis of CO₂ trapping capacities and long-term migration for geological formations in the Norwegian North Sea using MRST-co2lab. *Computers & Geoscience*, 79: 15–26, 2015. doi:[10.1016/j.cageo.2015.03.001](https://doi.org/10.1016/j.cageo.2015.03.001).
- [198] H. M. Nilsen, K.-A. Lie, O. Møyner, and O. Andersen. Spill-point analysis and structural trapping capacity in saline aquifers using MRST-co2lab. *Computers & Geoscience*, 75:33–43, 2015. doi:[10.1016/j.cageo.2014.11.002](https://doi.org/10.1016/j.cageo.2014.11.002).

- [199] H. M. Nilsen, J. M. Nordbotten, and X. Raynaud. Comparison between cell-centered and nodal based discretization schemes for linear elasticity. *arXiv preprint arXiv:1604.08410*, 2016.
- [200] J. Nordbotten, I. Aavatsmark, and G. Eigestad. Monotonicity of control volume methods. *Numer. Math.*, 106(2):255–288, 2007. doi:[10.1007/s00211-006-0060-z](https://doi.org/10.1007/s00211-006-0060-z).
- [201] J. M. Nordbotten. Convergence of a cell-centered finite volume discretization for linear elasticity. *SIAM J. Numer. Anal.*, 53(6):2605–2625, 2015. doi:[10.1137/140972792](https://doi.org/10.1137/140972792).
- [202] J. M. Nordbotten. Stable cell-centered finite volume discretization for biot equations. *SIAM J. Numer. Anal.*, 54(2):942–968, 2016. doi:[10.1137/15M1014280](https://doi.org/10.1137/15M1014280).
- [203] Y. Notay. An aggregation-based algebraic multigrid method. *Electron. Trans. Numer. Anal.*, 37:123–140, 2010.
- [204] A. S. Odeh. Comparison of solutions to a three-dimensional black-oil reservoir simulation problem. *J. Petrol. Techn.*, 33(1):13–25, 1981. doi:[10.2118/9723-PA](https://doi.org/10.2118/9723-PA).
- [205] P.-E. Øren, S. Bakke, and O. J. Arntzen. Extending predictive capabilities to network models. *SPE J.*, 3(4):324–336, 1998.
- [206] Oystein S. Klemetsdal, R. L. Berge, K.-A. Lie, H. M. Nilsen, and O. Møyner. Unstructured gridding and consistent discretizations for reservoirs with faults and complex wells. In *SPE Reservoir Simulation Conference, Montgomery, Texas, USA, 20-22 February 2017*, 2017. doi:[10.2118/182679-MS](https://doi.org/10.2118/182679-MS).
- [207] M. Pal, S. Lamine, K.-A. Lie, and S. Krogstad. Validation of the multi-scale mixed finite-element method. *Int. J. Numer. Meth. Fluids*, 77(4):206–223, 2015. doi:[10.1002/fld.3978](https://doi.org/10.1002/fld.3978).
- [208] H.-Y. Park and A. Datta-Gupta. Reservoir management using streamline-based flood efficiency maps and application to rate optimization. In *Proceedings of the SPE Western North American Region Meeting*, 7-11 May 2011, Anchorage, Alaska, USA, 2011. doi:[10.2118/144580-MS](https://doi.org/10.2118/144580-MS).
- [209] 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](https://doi.org/10.2118/10528-PA). SPE 10528-PA.
- [210] D. W. Peaceman. *Fundamentals of Numerical Reservoir Simulation*. Elsevier Science Inc., New York, NY, USA, 1991. ISBN 0444415785.
- [211] D. W. Peaceman et al. Interpretation of well-block pressures in numerical reservoir simulation. *Soc. Petrol. Eng. J.*, 18(3):183–194, 1978.
- [212] P.-O. Persson and G. Strang. A simple mesh generator in matlab. *SIAM Review*, 46(2):329–345, 2004. doi:[10.1137/S0036144503429121](https://doi.org/10.1137/S0036144503429121).
- [213] Ø. Pettersen. Basics of reservoir simulation with the eclipse reservoir simulator. Lecture notes, Department of Mathematics, University of

- Bergen, 2006. URL http://folk.uib.no/fciop/index_htm_files/ResSimNotes.pdf.
- [214] Ø. Pettersen. Horizontal simulation grids as alternative to structure-based grids for thin oil-zone problems: a comparison study on a Troll segment. *Comput. Geosci.*, 16(2):211–230, 2012. doi:[10.1007/s10596-011-9240-8](https://doi.org/10.1007/s10596-011-9240-8).
 - [215] G. F. Pinder and W. G. Gray. *Essentials of Multiphase Flow in Porous Media*. John Wiley & Sons, Hoboken, New Jersey, USA, 2008.
 - [216] D. W. Pollock. Semi-analytical computation of path lines for finite-difference models. *Ground Water*, 26(6):743–750, 1988.
 - [217] 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.
 - [218] T. C. Potempa. *Finite element methods for convection dominated transport problems*. PhD thesis, Rice University, 1982. URL <https://scholarship.rice.edu/handle/1911/15714>.
 - [219] M. Prevost, M. G. Edwards, and M. J. Blunt. Streamline tracing on curvilinear structured and unstructured grids. *SPE J.*, 7(2):139–148, June 2002.
 - [220] A. F. Rasmussen and K.-A. Lie. Discretization of flow diagnostics on stratigraphic and unstructured grids. In *ECMOR XIV – 14th European Conference on the Mathematics of Oil Recovery, Catania, Sicily, Italy, 8-11 September 2014*. EAGE, 2014. doi:[10.3997/2214-4609.20141844](https://doi.org/10.3997/2214-4609.20141844).
 - [221] 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.
 - [222] P. Renard and G. De Marsily. Calculating equivalent permeability: a review. *Adv. Water Resour.*, 20(5):253–278, 1997.
 - [223] L. A. Richards. Capillary conduction of liquids through porous mediums. *Journal of Applied Physics*, 1(5):318–333, 1931. doi:[10.1063/1.1745010](https://doi.org/10.1063/1.1745010).
 - [224] P. Samier. A finite element method for calculation transmissibilities in n-point difference equations using a non-diagonal permeability tensor. In Guérillot, editor, *2nd European Conference on the Mathematics of Oil Recovery*, pages 121–130. Editions TECHNIP, 1990.
 - [225] T. Sandve, I. Berre, and J. Nordbotten. An efficient multi-point flux approximation method for discrete fracturematrix simulations. *J. Comput. Phys.*, 231(9):3784 – 3800, 2012. doi:[10.1016/j.jcp.2012.01.023](https://doi.org/10.1016/j.jcp.2012.01.023).
 - [226] Eclipse. *ECLIPSE: Reference Manual*. Schlumberger, 2014.1 edition, 2014.
 - [227] Eclipse. *ECLIPSE reservoir simulation software: Technical description*. Schlumberger, 2014.1 edition, 2014.
 - [228] Eclipse. *ECLIPSE 100 User Course*. Schlumberger GeoQuest, 1999.

- [229] S. Shah, O. Møyner, M. Tene, K.-A. Lie, and H. Hajibeygi. The multi-scale restriction smoothed basis method for fractured porous media. *J. Comput. Phys.*, 318:36–57, 2016. doi:[10.1016/j.jcp.2016.05.001](https://doi.org/10.1016/j.jcp.2016.05.001).
- [230] M. Shahvali, B. Mallison, K. Wei, and H. Gross. An alternative to streamlines for flow diagnostics on structured and unstructured grids. *SPE J.*, 17(3):768–778, 2012. doi:[10.2118/146446-PA](https://doi.org/10.2118/146446-PA).
- [231] L. F. Shampine, R. Ketscher, and S. A. Forth. Using AD to solve BVPs in MATLAB. *ACM Trans. Math. Software*, 31(1):79–94, 2005.
- [232] G. Shook and K. Mitchell. A robust measure of heterogeneity for ranking earth models: The F-Phi curve and dynamic Lorenz coefficient. In *SPE Annual Technical Conference and Exhibition, 4-7 October, New Orleans, Louisiana*, 2009. doi:[10.2118/124625-MS](https://doi.org/10.2118/124625-MS).
- [233] C.-W. Shu. Total-variation-diminishing time discretizations. *SIAM J. Sci. Stat. Comput.*, 9(6):1073–1084, 1988. doi:[10.1137/0909073](https://doi.org/10.1137/0909073).
- [234] C.-W. Shu. *Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws*. Springer, 1998. doi:[10.1007/BFb0096355](https://doi.org/10.1007/BFb0096355).
- [235] G. R. Shubin and J. B. Bell. An analysis of the grid orientation effect in numerical simulation of miscible displacement. *Computer Methods in Applied Mechanics and Engineering*, 47(1):47–71, 1984. doi:[10.1016/0045-7825\(84\)90047-1](https://doi.org/10.1016/0045-7825(84)90047-1).
- [236] H. L. Stone. Probability model for estimating three-phase relative permeability. *J. Petrol. Tech.*, 22(02):214–218, 1970. doi:[10.2118/2116-PA](https://doi.org/10.2118/2116-PA).
- [237] H. L. Stone. Estimation of three-phase relative permeability and residual oil data. *J. Pet. Technol.*, 12(4), 1973. doi:[10.2118/73-04-06](https://doi.org/10.2118/73-04-06).
- [238] K. Stüben. A review of algebraic multigrid. *J. Comput. Appl. Math.*, 128(1):281–309, 2001.
- [239] Technische Universität Darmstadt. Automatic Differentiation for Matlab (ADiMat). URL <http://www.adimat.de/>. [Online; accessed 15-04-2014].
- [240] M. R. Thiele and R. P. Batycky. Water injection optimization using a streamline-based workflow. In *Proceedings of the SPE Annual Technical Conference and Exhibition*, 5-8 October 2003, Denver, Colorado, 2003. doi:[10.2118/84080-MS](https://doi.org/10.2118/84080-MS).
- [241] G. W. Thomas. *Principles of hydrocarbon reservoir simulation*. IHRDC, Boston, MA, Jan 1981.
- [242] M. R. Todd, P. M. O'Dell, and G. J. Hirasaki. Methods for increased accuracy in numerical reservoir simulators. *Society of Petroleum Engineers Journal*, 12(06):515–530, 1972. doi:[10.2118/3516-PA](https://doi.org/10.2118/3516-PA).
- [243] Tomlab Optimization Inc. Matlab Automatic Differentiation (MAD). URL <http://matlabad.com/>. [Online; accessed 15-04-2014].
- [244] S. M. Toor, M. G. Edwards, A. H. Dogru, and T. M. Shaalan. Boundary aligned grid generation in three dimensions and cvd-mpfa discretization. In *SPE Reservoir Simulation Symposium, 23-25 February, Houston, Texas, USA*, 2015. doi:[10.2118/173313-MS](https://doi.org/10.2118/173313-MS).

- [245] E. F. Toro. *Riemann solvers and numerical methods for fluid dynamics*. Springer-Verlag, Berlin, third edition, 2009. doi:[10.1007/b79761](https://doi.org/10.1007/b79761). A practical introduction.
- [246] J. A. Trangenstein. *Numerical solution of hyperbolic partial differential equations*. Cambridge University Press, Cambridge, 2009.
- [247] J. A. Trangenstein and J. B. Bell. Mathematical structure of the black-oil model for petroleum reservoir simulation. *SIAM J. Appl. Math.*, 49(3):749–783, 1989. ISSN 0036-1399.
- [248] E. Ucar, I. Berre, and E. Keilegavlen. Simulation of slip-induced permeability enhancement accounting for multiscale fractures. In *Fourtieth Workshop on Geothermal Reservoir Engineering, Stanford University, Stanford, California, January 26–28, 2015*, 2015.
- [249] M. T. van Genuchten. Closed-form equation for predicting the hydraulic conductivity of unsaturated soils. *Soil Science Soc. America J.*, 44(5):892–898, 1980. doi:[10.2136/sssaj1980.03615995004400050002x](https://doi.org/10.2136/sssaj1980.03615995004400050002x).
- [250] A. Verma. ADMAT: Automatic differentiation in MATLAB using object oriented methods. In *SIAM Interdisciplinary Workshop on Object Oriented Methods for Interoperability*, pages 174–183, 1999.
- [251] D. V. Voskov and H. A. Tchelepi. Comparison of nonlinear formulations for two-phase multi-component eos based simulation. *J. Petrol. Sci. Engr.*, 82-83(0):101–111, 2012. ISSN 0920-4105. doi:[10.1016/j.petrol.2011.10.012](https://doi.org/10.1016/j.petrol.2011.10.012).
- [252] D. V. Voskov, H. A. Tchelepi, and R. Younis. General nonlinear solution strategies for multiphase multicomponent eos based simulation. In *SPE Reservoir Simulation Symposium, 2–4 February, The Woodlands, Texas*, 2009. doi:[10.2118/118996-MS](https://doi.org/10.2118/118996-MS).
- [253] J. R. Wallis, R. P. Kendall, and T. E. Little. Constrained residual acceleration of conjugate residual methods. In *SPE Reservoir Simulation Symposium, 10-13 February, Dallas, Texas*, 1985. doi:[10.2118/13536-MS](https://doi.org/10.2118/13536-MS).
- [254] X. Wang and H. A. Tchelepi. Trust-region based solver for nonlinear transport in heterogeneous porous media. *Journal of Computational Physics*, 253:114–137, 2013. doi:[10.1016/j.jcp.2013.06.041](https://doi.org/10.1016/j.jcp.2013.06.041).
- [255] Y. Wang, H. Hajibeygi, and H. A. Tchelepi. Algebraic multiscale solver for flow in heterogeneous porous media. *J. Comput. Phys.*, 259:284–303, 2014. doi:[10.1016/j.jcp.2013.11.024](https://doi.org/10.1016/j.jcp.2013.11.024).
- [256] X.-H. Wen and J. J. Gómez-Hernández. Upscaling hydraulic conductivities in heterogeneous media: An overview. *J. Hydrol.*, (183):ix–xxxii, 1996.
- [257] J. A. Wheeler, M. F. Wheeler, and I. Yotov. Enhanced velocity mixed finite element methods for flow in multiblock domains. *Comput. Geosci.*, 6(3-4):315–332, 2002. doi:[10.1023/A:1021270509932](https://doi.org/10.1023/A:1021270509932).
- [258] M. F. Wheeler, T. Arbogast, S. Bryant, J. Eaton, Q. Lu, M. Peszynska, and I. Yotov. A parallel multiblock/multidomain approach for reservoir

- simulation. In *SPE Reservoir Simulation Symposium, 14-17 February, Houston, Texas*, pages 51–61, 1999. doi:[10.2118/51884-MS](#).
- [259] S. Whitaker. Flow in porous media I: A theoretical derivation of Darcy's law. *Transp. Porous Media*, 1(1):3–25, 1986. ISSN 1573-1634. doi:[10.1007/BF01036523](#).
- [260] C. H. Whitson and M. R. Brulé. *Phase behavior*. Number 20 in SPE Monograph Series. Society of Petroleum Engineers, Richardson, Texas, 2000.
- [261] O. Wiener. *Abhandlungen der Matematisch*. PhD thesis, Physischen Klasse der Königlichen Sächsischen Gesellschaft der Wissenschaften, 1912.
- [262] X. H. Wu, Y. Efendiev, and T. Y. Hou. Analysis of upscaling absolute permeability. *Discrete Contin. Dyn. Syst. Ser. B*, 2(2):185–204, 2002.
- [263] J. L. Yanosik and T. A. McCracken. A nine-point, finite-difference reservoir simulator for realistic prediction of adverse mobility ratio displacements. *Society of Petroleum Engineers Journal*, 19(04):253–262, 1979. doi:[10.2118/5734-PA](#).
- [264] R. Younis. *Advances in Modern Computational Methods for Nonlinear Problems; A Generic Efficient Automatic Differentiation Framework, and Nonlinear Solvers That Converge All The Time*. PhD thesis, Stanford University, Palo Alto, California, 2009.
- [265] R. Younis and K. Aziz. Parallel automatically differentiable data-types for next-generation simulator development. In *SPE Reservoir Simulation Symposium, 26–28 February, Houston, Texas, USA*, 2007. doi:[10.2118/106493-MS](#). SPE 106593-MS.
- [266] P. Zhang, G. E. Pickup, and M. A. Christie. A new upscaling approach for highly heterogenous reservoirs. In *SPE Reservoir Simulation Symposium, 31 January-2 February, The Woodlands, Texas*, SPE 93339, 2005.
- [267] Y. Zhou, H. A. Tchelepi, and B. T. Mallison. Automatic differentiation framework for compositional simulation on unstructured grids with multi-point discretization schemes. In *SPE Reservoir Simulation Symposium, 21-23 February, The Woodlands, Texas*, 2011. doi:[10.2118/141592-MS](#). SPE 141592-MS.