SPE 173317-MS

MRST-AD – an Open-Source Framework for Rapid Prototyping and Evaluation of Reservoir Simulation Problems

Stein Krogstad, Knut–Andreas Lie, Olav Møyner, Halvor Møll Nilsen, Xavier Raynaud, Bård Skaflestad, SINTEF ICT.

This paper was prepared for presentation at the SPE Reservoir Simulation Symposium held in Houston, Texas, USA, 23-25 February 2015.

Abstract

We present `MRST-AD`, a free, open-source framework written as part of the Matlab Reservoir Simulation Toolbox and designed to provide researchers with the means for rapid prototyping and experimentation for problems in reservoir simulation. The article outlines the design principles and programming techniques used and explains in detail the implementation of a full-featured, industry-standard black-oil model on unstructured grids. The resulting simulator has been thoroughly validated against a leading commercial simulator on benchmarks from the SPE Comparative Solution Projects, as well as on a real-field model (Voador, Brazil). We also show in detail how practitioners can easily extend the black-oil model with new constitutive relationships, or additional features such as polymer flooding, thermal and reactive effects, and immediately benefit from existing functionality such as constrained-pressure-residual (CPR) type preconditioning, sensitivities and adjoint-based gradients.

Technically, `MRST-AD` combines three key features: (i) a highly vectorized scripting language that enables the user to work with high-level mathematical objects and continue to develop a program while it runs; (ii) a flexible grid structure that enables simple construction of discrete differential operators; and (iii) automatic differentiation that 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. We have implemented a modular, efficient framework for implementing and comparing different physical models, discretizations, and solution strategies by combining imperative and object-oriented paradigms with functional programming. The toolbox also offers additional features such as upscaling and grid coarsening, consistent discretizations, multiscale solvers, flow diagnostics and interactive visualization.

Introduction

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? This is the main question that will be discussed in this paper. Our premise is that prototyping and validating new mathematical models and computational methods typically is painstakingly slow. There are many reasons for this. The first is that there is often a disconnect between the mathematical abstractions used to express new ideas and the constructs of the computer language used to realize the resulting computational algorithm. Typically, to achieve high computational performance of your new algorithm you end up spending most of your time on a low level working with loops and indices when working in a procedural language like FORTRAN or C. 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 (e.g., like expression templates). However, in our experience these advanced features are alien and unintuitive to researchers without extensive training in computer science. Even if you are familiar with such concepts and have made a framework with sufficient flexibility, you still face the never-ending frustration caused by different versions of compilers and (third-party) libraries that follow most compiled languages. All these problems are largely avoided by the use of a multi-paradigm numerical environment and fourth-generation programming language like Matlab or Python. Moreover, scripting languages enable the user to interactively analyse, change and extend data objects, include new features, while the program is being run. This feature is essential in a debugging process, when one tries to understand why a given numerical method fails to produce the results one excepts it to give. Altogether this enables a very versatile development process that is not easily matched by any compiled language.

A second problem that may impede rapid prototyping of new ideas is that in many cases you need to include ideas and algorithms developed by others to demonstrate the feasibility and validity of your own contribution. Unfortunately, there is often a long leap from reading a paper presenting a new method to having it work as part of your own code. The focus of journal papers is on novel scientific ideas rather than practical implementations. It is therefore quite common that authors unwittingly

fail to report minor details, implementation choices, and parameter settings that are seemingly unimportant for the scientific idea but turn out to be essential to get the method working in practice. As a result, you may end up spending endless hours trying to reverse-engineering an algorithm, only to discover that you cannot use the results reported in the original paper to verify your implementation because the numerical examples lack exact specification of parameters and underlying assumptions and therefore only cannot be easily reproduced. This problem is currently addressed by the *reproducible computational science*, which tries to bring out into the open the computer programs and the data that are required to reproduce the exact results discussed in a paper.

The third problem is that going from demonstrating the superiority of a new model or method on idealized, synthetic cases to providing a proper validation on complex problems encountered in the daily work of reservoir engineers usually is a significant undertaking. For real models, simulators include a large number of tests and branching, with adjustments which are necessary to handle the variety and complexity of input data. If you want to validate your new ideas on realistic test cases and cannot (or will not) use a commercial reservoir simulator, you typically end up spending a lot of time implementing functionality that has little to do with your idea and often is not well described in the literature. Another problem is, of course, that it is usually quite difficult for independent researchers to get access to good test problems that can be used to validate new ideas in realistic settings. Possible solutions to these problems include creating open community codes that can be used to read and process industry-standard input formats, as well as establishing repositories that provide open access to representative models and benchmark cases that cover a wide range of physical settings.

Over the past decade, we have been consistently working on improving ways to speed up our development and validation cycle for new models and algorithms. As a result of this, we have developed the Matlab Reservoir Simulation Toolbox (MRST 2014), which is a toolbox for rapid prototyping of new models and computational methods written using the high-level Matlab scripting language. To enable other researchers to leverage our work, MRST has been published as free open-source code since 2009 and has been downloaded and used by hundreds of users worldwide. Obviously, the software does not solve all the problems outlined above, but we think it is a good start. In the following we will therefore present some of the technical ideas that we believe may be useful to others.

At a first glance, it may seem strange to use Matlab to develop simulation methods for complex 3D reservoir models. Our aim with this paper is to hopefully convince the reader that this is not so. First of all, Matlab offers a vectorized syntax and a wide range of mathematical functions that can be used to create very compact programs: In (Aarnes et al. 2007a) we demonstrated how to write a two-phase, incompressible simulator on 3D, Cartesian grids in approximately fifty lines of code that mostly amounted to setting model parameters and manipulating vectors and matrices using vectorized index operations. (See also (Alberty et al. 1999) for a similar demonstration for finite-element methods). Secondly, Matlab provides robust and efficient implementation of a large number of numerical algorithms that are highly useful when developing new computational methods.

For small systems, an interpreted scripting language may introduce a significant computational overhead, but this is by far out-weighted by the very flexible and efficient development process enabled by such a language. At any point in the execution of a Matlab program, you can stop the program to inspect your data, modify their values, add or remove new fields in your data structures[1], introduce new data, execute any number of statements and function calls, and go back and reiterate parts of the program, possibly with modified or additional data. Not only does this make debugging quite simple and efficient, but it also enables you to dynamically develop your algorithm by modifying your code or adding new functionality as the algorithm is being run. For large systems, most of the computational time should ideally be spent processing floating-point numbers and here Matlab is surprisingly efficient and fully comparable with compiled languages. Using a single core on a standard workstation with 24 GiB of memory, we were, for instance, able to solve a single-phase pressure equation with slightly more than sixteen million grid cells represented in a fully unstructured format within approximately one minute.

In this article we will outline and discuss how MRST can be used to simulate black-oil type models seen in industry-standard reservoir simulation. Most commercial reservoir simulators are based on fully implicit formulations to ensure robustness over a wide range of models and flow scenarios. We have implemented such formulations in our software using automatic differentiation to compute the Jacobi matrices required for the nonlinear Newton-type solver. Combined with vectorization and suitable abstract operators for spatial discretizations, this implies that models can be written in a compact form that is close to the corresponding mathematical formulation. It is therefore simple to implement new models: you implement the model equations and then the software generates the discretizations and linearizations needed to obtain a working simulator. The framework also includes state-of-the-art methods for time-step control, preconditioning methods (CPR) and algebraic multigrid solvers for high numerical efficiency, as well as support for reading and parsing industry-standard input decks describing the reservoir, the wells, and the simulation schedule.

Quick Overview of the Software

The software consists of two parts, see **Fig. 1**. The relatively slim core module contains data structures and basic routines necessary to set up, solve, and visualize incompressible single and two-phase models on structured and unstructured grids. Major parts of the core module were first presented in (Lie et al. 2012), including data structures for representing grids, petrophysical parameters, simple fluid models, wells, and boundary conditions along with a discussion of consistent discretizations on general polyhedral grids. The same material is discussed more thoroughly by Lie (2014), who also explains how to use the functionality

[1]This is not possible in a compiled language unless you explicitly have told your program about the possible existence of such a data field upfront

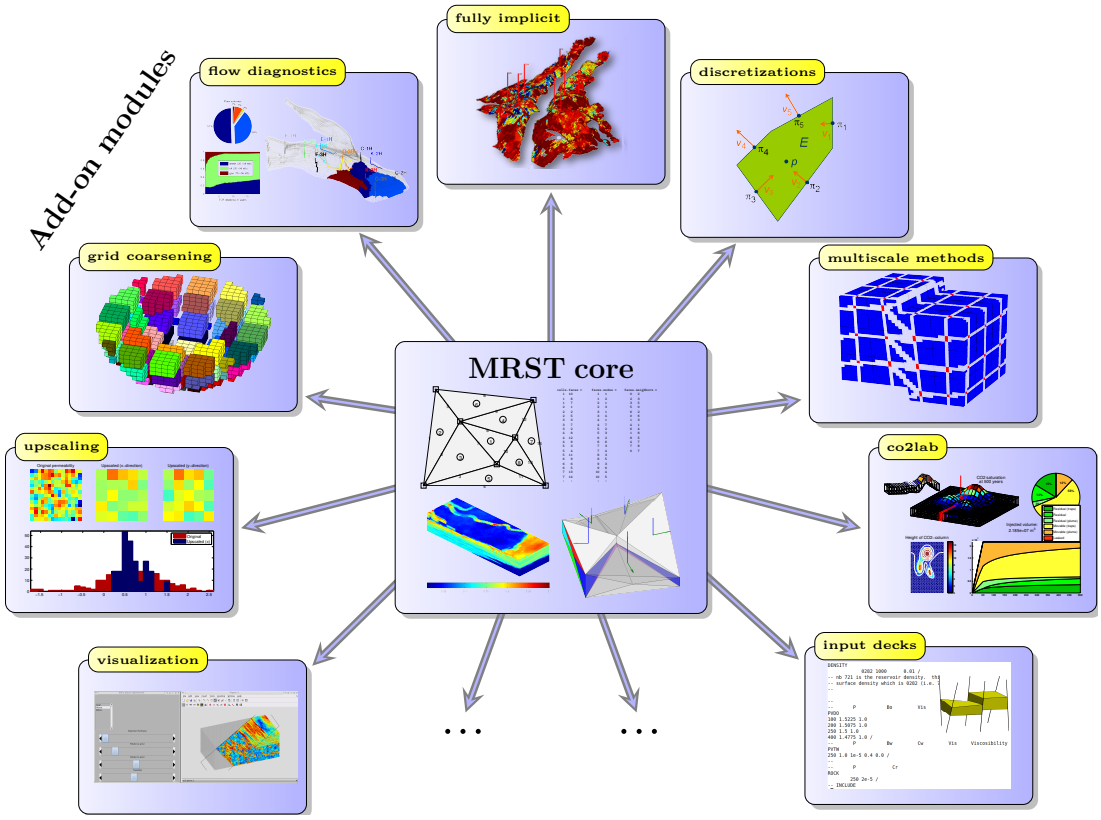

Fig. 1—Organization of the Matlab Reservoir Simulation Toolbox (MRST) into a core module that provides basic data structures and simplified solvers, and a set of add-on modules offering more advanced models, solvers, viewers, and workflow tools.

for automatic differentiation in `MRST-core` to implement nonlinear, single-phase pressure solvers and outlines various methods for grid coarsening. Routines in `MRST-core` have been quite stable over many years and are generally well documented in a format that follows the MATLAB standard.

The second part of the software consists of a set of add-on modules that extend, complement, and override existing features from `MRST-core`, typically in the form of specialized or more advanced solvers like consistent discretizations (Lie et al. 2012; Nilsen et al. 2012) or workflow tools like grid coarsening and standard methods for single and two-phase upscaling. Others modules offer convenient functionality like reading and processing of industry-standard input decks, interactive visualization, C-acceleration of selected routines from `MRST-core`, etc. These modules are robust, well-documented, and contain features that are reasonably interoperable and will likely not change in future releases, and could therefore have been included in the core module had we not decided to keep it as small as possible.

The remaining and workflow tools, on the other hand, are constantly changing to support ongoing research. This includes methods for coarsening grids to adapt to geology and flow features (Aarnes et al. 2007b; Hauge and Aarnes 2009; Hauge et al. 2012; Lie et al. 2014a), upscaling methods (Raynaud et al. 2014; Hilden et al. 2014), and various multiscale methods including mixed finite-elements (Aarnes et al. 2006, 2008; Alpak et al. 2012; Krogstad et al. 2012), finite-volume methods (Møyner and Lie 2014a,b), and POD-based model reduction (Krogstad 2011). We also work on flow diagnostics methods (Møyner et al. 2014) that can be used to establish connections and basic volume estimates and quickly provide a qualitative picture of the flow patterns in the reservoir, methods for estimating trapping capacity and containment in geological CO_2 storage (Andersen et al. 2014; Lie et al. 2014b), and various adjoint formulations for production optimization (Krogstad et al. 2011; Raynaud et al. 2014). All these modules are publicly available from the software's webpage (MRST 2014). In addition, a few third-party modules have also been developed, including a module for ensemble Kalman filter (EnKF) methods (Leeuwenburgh et al. 2011) and one for discrete-fracture systems (Sandve et al. 2012).

`MRST-core`: Grids, Single-Phase Flow, and Vectorization

To understand the capability for rapid prototyping of fully-implicit simulators, we must first briefly explain the basic functionality for grids and discretizations that is implemented in `MRST-core`. A much more thorough introduction can be found in (Lie 2014).

When working with grids that are more complex than simple box models–like the stratigraphic corner-point, PEBI, or cut-cell formats seen in most contemporary field models–one needs to introduce some kind of data structure to represent the grid.

In many simulators, the actual geometry grid is only seen by the preprocessor, which constructs a connection graph with cell volumes and cell properties associated with the vertices and inter-cell transmissibilities and fluxes associated with the edges. In our software, however, we have chosen to keep the grid and the petrophysical properties as fundamental and separate entities that are present throughout the simulation. In particular, the grid object is passed as input to almost all solvers and visualization routines. To ensure interoperability among a wide variety of different grid types and computational methods, we have chosen to use a relatively rich format for the grid object. In this format, grids are assumed to consist of a set of matching polygonal cells, which are represented using three fields: `cells`, `faces`, and `nodes`. Each of the n_c cells corresponds to a subset of the n_f faces, and each face to a set of edges, which again are determined by the nodes. To define the topology of the grid, we use two mappings. The first is given by $F : \{1, \ldots, n_c\} \to \{0, 1\}^{n_f}$ and maps a cell to the set of faces that delimit this cell. In the grid structure `G` in MRST, this is represented as an array called `G.cells.faces`, in which the first column that gives the cell numbers is not stored since it is redundant and instead must be computed by a call `gridCellNo(G)`. The second mapping brings you from a given face to the two neighboring cells, $N_1, N_2 : \{1, \ldots, n_f\} \to \{0, \ldots, n_c\}$, where 0 has been included to denote the exterior of the computational domain. In `G`, N_1 is given by `G.faces.neighbors(:,1)` and N_2 by `G.faces.neighbors(:,2)`. The `cell` and `face` objects also contain geometrical properties like centroids, volumes, areas, and normal vectors.

For many structured grid types, the grid object outlined above will obviously contain a lot of redundant information. However, rather than simplifying the grid object in these cases by removing redundant information, we have chosen to extend the grid object with extra information (like ijk or ik numbering) that can be exploited when present. Likewise, many algorithms involve the use of grid coarsening. In MRST, all coarse grids are assumed to be partitions of an underlying fine grid, i.e., to be defined entirely by a partition vector p whose element $p(i) = j$ if fine cell i belongs to coarse block j. Coarse grids can be represented in the same rich format as outlined above and used interchangeably with fine grids by most of the standard solvers in MRST. The grid structure is discussed at length in (Lie 2014) along with detailed descriptions of how to construct such grids from an input file, using one of the many grid-factory routines that come with the software, or by writing your own grid-generation script.

To implement computational algorithms on the grid, we use vectorized index operations in combination with the mappings between cells and faces outline above. Conceptually, this may appear more complicated than writing for-loops based on explicit counting in structured topographies. In our experience, however, the extra effort needed to understand these abstractions is by far out-weighted by the advantage of being able to effortlessly switch between various grid formats. To be more specific, we consider the following incompressible, single-phase flow problem

$$\nabla \cdot \vec{v} = q, \qquad \vec{v} = -\mathbf{K}\nabla p, \qquad \vec{x} \in \Omega \subset \mathbb{R}^3, \tag{1}$$

discretized by a standard two-point finite-volume approximation. Here, $\vec{v}$ is the Darcy velocity, $\mathbf{K}$ is the absolute permeability, p is the fluid pressure, and q is a volumetric source. For brevity, we will not use a well model, assume that $\mathbf{K}$ is isotropic, and only consider no-flow boundary conditions on $\partial\Omega$. **Fig. 2** highlights the essential parts of an implementation that is valid for 2D polygonal and 3D polyhedral grids and shows the close correspondence between the mathematical description of the computational method and corresponding Matlab statements. Adding anisotropic effects and extending the pressure solver to multiple phases is straightforward, but accounting for multipliers, wells, and general boundary conditions complicates the construction of the discrete system. In MRST, the computation of one-sided transmissibilities and the assembly and solution of the discrete linear system are therefore implemented as two functions `hT=computeTrans(G,rock)` and `state=incompTPFA(state, G, hT, fluid)`, where `state` is a data object that contains the reservoir state (pressure, flux, saturation, etc) and `fluid` is a data object that contains the fluid model (viscosity, density, relative permeability, etc). This type of imperative, vectorized programming is used in all solvers that are part of the core module, as well as in all other incompressible solvers and simulators that rely on a sequential splitting of flow and transport. In particular, our implementation of mimetic and MPFA discretizations can be interfaced through similar functional calls.

Differential Operators and Automatic Differentiation

While the double-index notation $T_{i,k}$ and T_{ik} used above should be easy to comprehend, its usage becomes more involved when we want to discretize more complex equations than the Poisson equation. We therefore introduce discrete operators `div` and `grad` for the divergence and gradient operators, respectively, to enable us to write the discretized equations in almost the same form as the continuous equations. The main advantage, however of introducing these operators is that they can be represented by sparse matrices in Matlab and their action computed by efficient matrix-vector multiplications.

The `div` operator is a linear mapping from faces to cells. Let $\boldsymbol{v} \in \mathbb{R}^{n_f}$ denote a discrete flux and $\boldsymbol{v}[f]$ its restriction onto face f with orientation from $N_1(f)$ to $N_2(f)$. The divergence of the flux restricted to cell c is given as

$$\texttt{div}(\boldsymbol{v})[c] = \sum_{f \in F(c)} \boldsymbol{v}[f]\, \mathbf{1}_{\{c=N_1(f)\}} - \sum_{f \in F(c)} \boldsymbol{v}[f]\, \mathbf{1}_{\{c=N_2(f)\}}, \tag{2}$$

where $\mathbf{1}_{\{expr\}}$ equals one when $expr$ is true and zero otherwise. Likewise, the `grad` operator maps from cell pairs to faces and restricted to face f it is defined as

$$\texttt{grad}(\boldsymbol{p})[f] = \boldsymbol{p}[N_2(f)] - \boldsymbol{p}[N_1(f)], \tag{3}$$

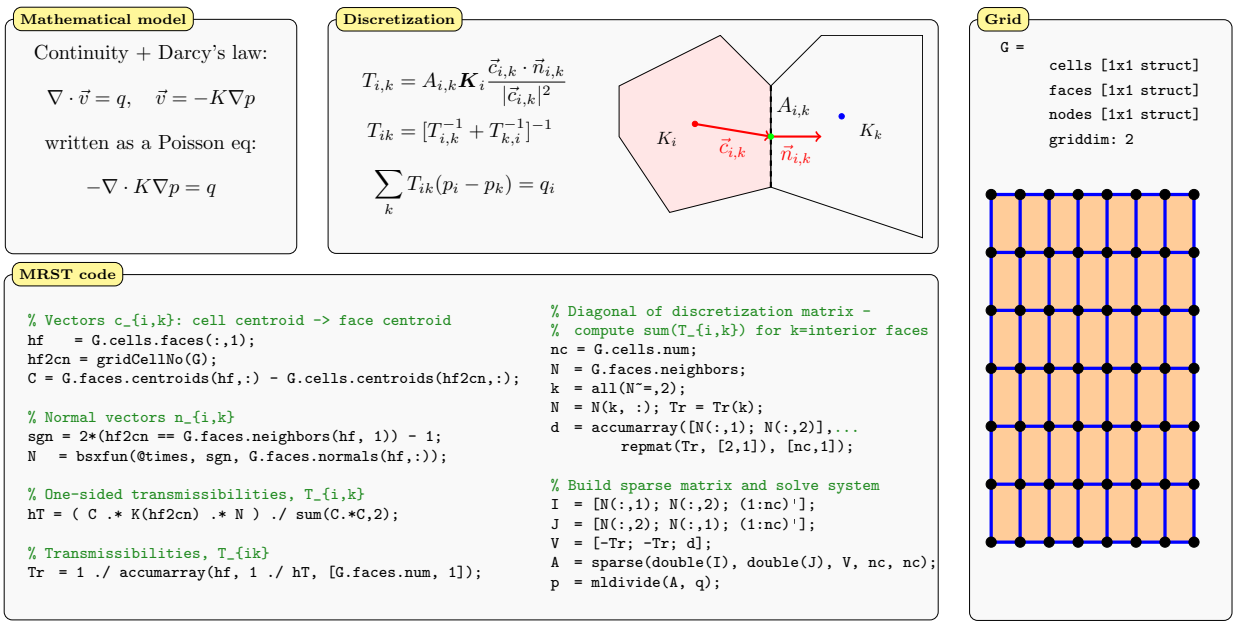

Fig. 2—Implementation of a two-point scheme for single-phase, incompressible flow using imperative programming. In the grid, the faces $A_{i,k}$ are referred to as *half faces* since they are associated with a particular grid cell K_i and a normal vector $\vec{n}_{i,k}$. Because the grid is matching, each interior half face will have a twin half face $A_{k,i}$ with identical area but opposite normal vector $\vec{n}_{k,i} = -\vec{n}_{i,k}$. The array `G.cells.faces` maps from cells to half faces, whereas `G.faces.neighbors` maps each face to two unique cell indices (one of the two is zero for a boundary face). Finally, the Matlab function `accumarray(subs,val)` is used in lieu of a for-loop and collects and sums all elements of `val` that correspond to identical values of `subs`.

for any $\boldsymbol{p} \in \mathbb{R}^{n_c}$. If we let $\boldsymbol{T}[f]$ denote the transmissibility of face f, we can then write the discrete version of Eq. 1 as

$$\mathtt{div}(\boldsymbol{v}) = \boldsymbol{q}, \qquad \boldsymbol{v} = -\boldsymbol{T}\mathtt{grad}(\boldsymbol{p}). \tag{4}$$

To demonstrate the utility of this abstraction, we consider a compressible single-phase pressure equation

$$\frac{\partial}{\partial t}(\phi\rho) + \nabla \cdot (\rho\vec{v}) = q, \qquad \vec{v} = -\frac{\mathbf{K}}{\mu}\left(\nabla p - g\rho\nabla z\right). \tag{5}$$

The primary unknown is the fluid pressure p and additional equations are supplied to provide relations between p and the other quantities, e.g., by specifying the porosity $\phi = \phi(p)$ as a function of p and an equation-of-state $\rho = \rho(p)$ for the density of the fluid. The fluid viscosity μ is assumed to be constant. Using the discrete operators introduced above, the basic implicit discretization of Eq. 5 reads,

$$\begin{aligned} \frac{(\boldsymbol{\phi\rho})^{n+1} - (\boldsymbol{\phi\rho})^n}{\Delta t^n} + \mathtt{div}(\boldsymbol{\rho v})^{n+1} &= \boldsymbol{q}^{n+1}, \\ \boldsymbol{v}^{n+1} = -\frac{\boldsymbol{K}}{\boldsymbol{\mu}}\left[\mathtt{grad}(\boldsymbol{p}^{n+1}) - g\boldsymbol{\rho}^{n+1}\mathtt{grad}(\boldsymbol{z})\right]&. \end{aligned} \tag{6}$$

Here, $\boldsymbol{\phi} \in \mathbb{R}^{n_c}$ denotes the vector porosity values per cell, $\boldsymbol{v}$ the vector of fluxes per face, and so on, and the superscript refers to discrete times at which one wishes to compute the unknown reservoir states and Δt denotes the time between two consecutive points in time.

When ϕ and ρ depend nonlinearly on p, we obtain a (highly) nonlinear system of equations that needs to be solved at each time step

$$\boldsymbol{G}(\boldsymbol{p}^{n+1}; \boldsymbol{p}^n) = \boldsymbol{0}. \tag{7}$$

Nonlinear systems of discrete equations arising from the discretization of (partial) differential equations are typically solved by Newton's method, which reads

$$\frac{\partial \boldsymbol{G}(\boldsymbol{p}^i)}{\partial \boldsymbol{p}^i}\delta\boldsymbol{p}^{i+1} = -\boldsymbol{G}(\boldsymbol{p}^i), \quad \boldsymbol{p}^{i+1} \leftarrow \boldsymbol{p}^i + \delta\boldsymbol{p}^{i+1}. \tag{8}$$

Here, $\boldsymbol{J}(\boldsymbol{p}^i) = \partial\boldsymbol{G}(\boldsymbol{p}^i)/\partial\boldsymbol{p}^i$ is the Jacobian matrix, while we refer to $\delta\boldsymbol{p}^{i+1}$ as the *Newton update* at iteration step number $i+1$. The Newton process will under certain smoothness and differentiability requirements exhibit quadratic convergence, provided

that one can obtain a sufficiently accurate Jacobian matrix. If $\boldsymbol{G}$ represents a set of complex equations, analytical derivation and subsequent implementation of the Jacobian can be both time-consuming and highly error prone and is often a bottleneck when implementing new mathematical models. Moreover, when you implement a new model or develop a new simulation algorithm, you will often want to experiment with different linearizations, i.e., vary which quantities are evaluated at step n and which are considered unknown at step $n+1$.

Automatic or algorithmic differentiation (Neidinger 2010) 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 and evaluation of simple functions. The key idea is to keep track of variables and their derivatives simultaneously; every time an operation is applied to a variable, the appropriate differential operation is applied to its derivative. Consider a scalar primary variable x and a function $f = f(x)$, whose AD-representations are the pairs $\langle x, 1\rangle$ and $\langle f, f_x\rangle$, where f_x is the derivative of f with respect to x. We now define the action of elementary operations and functions for all such pairs. As an example, addition and multiplication become

$$\langle f, f_x\rangle + \langle g, g_x\rangle = \langle f + g, f_x + g_x\rangle\,,$$
$$\langle f, f_x\rangle * \langle g, g_x\rangle = \langle fg, fg_x + f_x g\rangle\,.$$

In addition, one needs to use the chain rule to accumulate derivatives: if $f(x) = g(h(x))$, then $f_x(x) = \frac{dg}{dh}h_x(x)$. To implement AD in Matlab, we will use operator overloading. When Matlab encounters an expression of the form `a+b`, the software will choose one out of several different addition functions depending on the data types of `a` and `b`. All we therefore have to do is to introduce new addition functions for the various classes of data types that `a` and `b` may belong to. A nice introduction to how this is done is given by Neidinger (2010).

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 (DeBaun et al. 2005), but has mainly been pioneered through a reimplementation of the GPRS research simulator (Cao 2002). The new simulator, called AD-GPRS is primarily based on fully implicit formulations (Voskov et al. 2009; Zhou et al. 2011; Voskov and Tchelepi 2012) 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++ (Younis and Aziz 2007; Younis 2009). This way, the Jacobi matrices needed in the nonlinear Newton-type iterations can be constructed from the derivatives that are implicitly computed from when evaluating the residual equations. In (Li and Zhang 2014), the authors discuss how to use the alternative backward-mode differentiation to improve computational efficiency.

Automatic differentiation in MRST is implemented using operator-overloading in user-defined classes and relies on a relatively simple forward accumulation. The implementation 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, we use 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 is twofold: computational performance and user utility. A reservoir simulation model will in most cases consist of several equations (continuum equations, Darcy's law, equations of state, other constitutive relationships, control equations for wells, etc) that have different characteristics and play different roles in the overall equation system. Although we are using fully implicit discretizations in which one seeks to solve for all state variables simultaneously, we may still want to manipulate parts of the full equation system that e.g., represent 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, the derivatives with respect to different primary variables are represented as a list of matrices.

Automatic differentiation introduces 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. Moreover, user-defined classes are fairly new in MATLAB and there is still some overhead in using class objects and accessing their properties 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. A word of caution at the end: while `for`-loops in many cases will be quite efficient in Matlab (contrary to what is common belief), one should try to avoid loops that call AD for scalars and short arrays. Our AD class 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.

We now have all the necessary tools to implement a fully-implicit simulator for the nonlinear pressure equation Eq. 5. **Fig. 3** highlights the key parts of the implementation: creation of discrete operators, specification of constitutive functions, specification of discrete equations, and setup and solution of the linearized system inside the time loop. Wells are described using a standard Peaceman type well model. Notice that the control on the bottom-hole pressure is imposed as a separate equation. A complete example, including all code lines necessary to generate grid, petrophysical parameters, and well positions, is given in (Lie 2014).

The observant reader will notice that we tacitly have extended our programming model by using *anonymous* functions to compute constitutive relationships and evaluate the residual form of the discrete equations. An anonymous function is a function that is not stored in a program file, but is associated with a variable of type function handle. Anonymous functions can accept inputs and return outputs, just as standard functions, but can contain only a single executable statement. Such functions are used a lot in MRST, e.g., to define a new function $h(x, w)$ if you have a function $f(x, y, z, w, \dots)$ and only x and w varies in your algorithm.

Constitutive laws

```
cr   = 1e-6/barsa;
pr   = 200*barsa;
pvr  = poreVolume(G, rock);
pv   = @(p) pvr .* exp( cr * (p - pr) );
:
rho  = @(p) rhor .* exp( c * (p - pr) );
```

Initialization

```
z    = G.cells.centroids(:,3);
g    = norm(gravity);
veq  = ode23(@(z,p) g.*rho(p),[0, max(z)],pr);
pin  = reshape(deval(veq, z), [], 1);
```

Discrete operators

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

Pressure equation

```
dz  = grad(z);
v   = @(p) -(Tr/mu).*(grad(p) - g*avg(rho(p)).*dz);

pEq = @(p,p0,dt) (1/dt)*(pv(p).*rho(p) - pv(p0).*rho(p0)) ...
                        + div( avg(rho(p)).*v(p) );
```

Well equations

```
wc = W(1).cells; % connection grid cells
WI = W(1).WI;   % well-indices
dz = W(1).dZ;   % depth relative to bottom-hole

pcon = @(bhp)  bhp + g*dz.*rho(bhp); %connection pressures
qcon = @(p,bhp) WI .* (rho(p(wc))/mu).*(pcon(bhp) - p(wc));

rEq = @(p, bhp, qS) qS - sum(qcon(p, bhp))/rhoS;
cEq = @(bhp) bhp - 100*barsa;
```

Time loop

```
[p, bhp, qS] = initVariablesADI(pin, pin(wc(1))),0);
[pIx, bIx, qIx] = deal(1:nc, nc+1, nc+2);
t = 0;
while t < totTime,
   t = t + dt;
   resNorm = 1e99;
   p0  = double(p);
   nit = 0;
   while (resNorm > tol) && (nit <= maxits)
      eq1  = pEq(p, p0, dt);
      eq1(wc) = eq1(wc) - qcon(p, bhp);
      eqs = {eq1, rEq(p, bhp, qS), cEq(bhp)};
      eq  = cat(eqs{:});

      J   = eq.jac{1};
      upd = - (J \ eq.val);

      p.val   = p.val   + upd(pIx);
      bhp.val = bhp.val + upd(bIx);
      qS.val  = qS.val  + upd(qIx);

      resNorm = norm(eq.val);
      nit     = nit + 1;
   end
end
```

Fig. 3—Implementation of a two-point scheme to solve single-phase compressible flow using discrete differential operators, automatic differentiation, and a combination of imperative and functional programming. Refer to Fig. 2 for definition of grid mappings, transmissibility calculation, etc.

Arithmetic

```
mu = @(p)  mu0*(1+c_mu*(p-pr));
v  = @(p)  -(T./mu(avg(p))).*(grad(p) - g*avg(rho(p)).*dz);

qcon = @(p,bhp) WI.*(rho(p(wc))./mu(p(wc))).*(pcon(bhp)-p(wc));
```

Harmonic

```
hf2f  = sparse( double(hf), (1:numel(hf2cn))', 1);
hf2if = hf2f(all(N~=0,2), :);
fmob  = @(mu, p) 1./( hf2if*(mu(p(hf2cn))./hT) );
v = @(p)  -fmob(mu,p).*(grad(p) - g*avg(rho(p)).*dz);
```

Fig. 4—Arithmetic and harmonic averaging of a pressure-dependent viscosity for a compressible, single-phase flow equation. The minor extensions of the code shown in Fig. 3 are marked in red.

Rapid Prototyping

The main advantage of using abstract operators and automatic differentiation, as seen in the previous section, is that it localizes the implementation of the discrete model equations and that one avoids having to compute the various components of the Jacobian matrix by hand. This means that it is quite simple to extend the flow models with more physical effects, as we will see next.

Pressure-dependent viscosity. In the model discussed above, the viscosity was assumed to be constant, but it the general case it will increase with pressure, which may induce significant effects inside the reservoir. As an example, we consider a linear relationship, $\mu(p) = \mu_0[1 + c_\mu(p - p_r)]$. This requires changes in two parts of our discretization: the approximation of the Darcy flux across cell faces and the flow rate through a well connection. For the latter, we simply evaluate the viscosity using the pressure that was used to evaluate the density. For the Darcy flux, we have two choices: either use a simple arithmetic average, or replace the quotient of the transmissibility and the face viscosity by the harmonic average of the mobility $\lambda = \mathbf{K}/\mu$ in the adjacent cells. Both choices will lead to changes in the structure of the discrete nonlinear system. However, because we are using automatic differentiation, all we have to do is to reimplement the evaluation of the discrete equations as shown in **Fig. 4**. Here, `hf2if` represents a map from half faces with which the one-sided transmissibilities are associated to faces that are shared by two cells. Hence, premultiplying a vector of half-face quantities by `hf2if` amounts to summing the contributions from cells $N_1(f)$ and $N_2(f)$ for each face f. And this is it! You do not need to think of deriving a new Jacobi matrix–this is taken care of by automatic differentiation.

Thermal effects. As another example of rapid prototyping, we will extend the simple single-phase, compressible flow model Eq. 5 introduced above to account for thermal effects. That is, we extend our model to also include conservation on energy,

$$\begin{aligned} \frac{\partial}{\partial t}\left[\phi\rho(p,T)\right] + \nabla\cdot\left[\rho(p,T)\vec{v}\right] &= q, \qquad \vec{v} = -\frac{\mathbf{K}}{\mu(p,T)}\left[\nabla p - g\rho(p,T)\nabla z\right] \\ \frac{\partial}{\partial t}\left[\phi\rho(p,T)E_f(p,t) + (1-\phi)E_r(p,T)\right] + \nabla\cdot\left[\rho(p,T)H_f(p,T)\vec{v}\right] - \nabla\cdot\left[\boldsymbol{\kappa}\nabla T\right] &= q_e \end{aligned} \tag{9}$$

Here, the rock and the fluid are assumed to be in local thermal equilibrium, and T is temperature, E_f is the energy density per mass of the fluid, $H_f = E_f + p/\rho$ is the enthalpy density per mass, E_r is the energy per volume of the rock, and κ is the heat

Constitutive laws and operators

```
pvr  = poreVolume(G, rock);
pv   = @(p) pvr .* exp( cr * (p - pr) );
spv  = @(p) G.cells.volumes - pv(p);
:
rho  = @(p,T) rhor.*(1+(cp*(p - pr))).*exp(-ct*(T-Tr));
mu   = @(p,T) mu0*(1+cmup*(p-p_r)).*exp(-cmut*(T-T_r));
:
Hf   = @(p,T) Cw*T + (1-Tr*ct).*(p-pr)./rho(p,T);
Ef   = @(p,T) Hf(p,T) - p./rho(p,T);
Er   = @(T)   Cr*T;
:
upw  = @(x,flag) x(N(:,1)).*double(flag) ...
             + x(N(:,2)).*double(~flag);
```

Discrete equations

```
v   = @(p,T) -(Tr./mu(avg(p),avg(T))) ...
               .*( grad(p) - g*avg(rho(p,T)).*dz );

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) );

hEq = @(p, T, p0, T0, dt) ...
      (1/dt)*( pv(p).*rho(p,T).*Ef(p,T) + spv(p).*Er(T)
            - pv(p0).*rho(p0,T0).*Ef(p0,T0) - spv(p0).*Er(T0)) ...
      + div( upw(Hf(p,T),v(p,T)>0).*avg(rho(p,T)).*v(p,T) )...
      + div( -Th.*grad(T));
```

Fig. 5—Excerpts of the code necessary to extend the two-point simulator outlined in Fig. 3 to account for thermal effects. The transmissibility `Th` for the heat conductivity is computed in exact the same way as shown in Fig. 2 the rock transmissibility `Tr` with the permeability `K` replaced by the heat conductivity coefficient κ.

conduction coefficient of the rock. The primary variables are the fluid pressure and the temperature. We do not discuss the details of the new constitutive relationships except for noting 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 physically meaningful solutions.

To extend the simulator outlined in Fig. 3, we need to modify ρ, μ, and the existing discrete equations to include temperature dependence and introduce E_f, E_r, and H_f as anonymous functions and develop a discretization of the second conservation equation in Eq. 9. The accumulation and the heat-conduction terms are discretized in the same was as for the first conservation equation. For the second term, however, we need to introduce an upwind evaluation of the enthalpy density,

$$\mathtt{upw}(\boldsymbol{H}_f)[f] = \begin{cases} \boldsymbol{H}_f[N_1(f)], & \text{if } \boldsymbol{v}[f] > 0, \\ \boldsymbol{H}_f[N_2(f)], & \text{otherwise.} \end{cases} \tag{10}$$

Fig. 5 shows key parts of the corresponding code. In addition, there are trivial changes to the iteration loop to declare the correct variables as AD structures, evaluate the discrete equations and collect their residuals, and update the state variables. One must also make sure that heat sources are evaluated correctly for injection and production wells. These details, however, have been left out for brevity.

The observant reader will quickly realize that the code excerpts shown in Fig. 5 contains a number of redundant function evaluations: In each nonlinear iteration we keep re-valuating quantities that depend on `p0` and `T0` even though these stay constant for each time step. This can easily be avoided by moving the definition of the anonymous functions that evaluate the residual equations inside the outer time loop, see Fig. 3. Because each residual equation is defined as an anonymous function, we also observe that `v(p,T)` will be evaluated three times for each residual evaluation, once in `pEq` and twice in `hEq`, which means that `mu(avg(p),avg(T))` is evaluated three times and `rho(p,T)` is evaluated *seven* times, and so on. To cure this problem, we can move the computations of residuals inside a function in which the constitutive relationships can be computed one by one and stored in temporary variables, as discussed in the next section. 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.

Black-oil Type Models: Object-Orientation

The techniques outlined above are basically all you need to efficiently implement fully-implicit simulators. The most widely used fluid model in reservoir simulation is the black-oil family, which contains three components and the three phases water, oil, and gas. The black-oil equations can be written on the form

$$\begin{aligned} \partial_t(\phi b_o s_o) + \nabla \cdot (b_o \vec{v}_o) - b_o q_o &= 0, \\ \partial_t(\phi b_w s_w) + \nabla \cdot (b_w \vec{v}_w) - b_w q_w &= 0, \\ \partial_t[\phi(b_g s_g + b_o r_s s_o)] + \nabla \cdot (b_g \vec{v}_g + b_o r_s \vec{v}_o) - (b_g q_g + b_o r_s q_o) &= 0, \\ \vec{v}_\alpha = -(k_{r\alpha}/\mu_\alpha)\mathbf{K}(\nabla p_\alpha - \rho_\alpha g \nabla z), \qquad \alpha &= o, w, g. \end{aligned} \tag{11}$$

Here, s_α denotes the saturation, p_α the phase pressure, b_α the inverse formation volume factor (ratio between volume at elevated pressure and volume at surface conditions), $k_{r\alpha}$ is the relative permeability, and μ_α the viscosity of phase α. The gas is miscible in oil and the gas-oil ratio r_s for a saturated oil is a function of pressure. The formation volume factor and the viscosity of oil depend on pressure and the gas-oil ratio.

Operators

```
% N: interior faces -> adjacent cells

% Average at face
M = sparse((1:nf)'*[1 1],N,.5*ones(nf,2),nf,nc);
S.faceAvg = @(x) M*x;

% Div and grad
C  = sparse([(1:nf)'; (1:nf)'], N, ...
           ones(nf,1)*[1 -1], nf, G.cells.num);
S.Grad = @(x) -C*x;
S.Div  = @(x) C'*x;

% Upstream weighting
S.faceUpstr = @(flag, x) faceUpstr(flag, x, N, [nf, nc]);

function xu = faceUpstr(flag, x, N, sz)
upCell      = N(:,2);
upCell(flag) = N(flag,1);
xu = sparse((1:sz(1))', upCell, 1, sz(1), sz(2))*x;
```

Discrete equations

```
function [eqs,..] = eqsfiOW(state0, state, ..)
p  = state.pressure; sW = state.s(:,1);  % unknowns
p0 = state0.pressure; sW0 = state0.s(:,1); % previous step
:
% water properties (evaluated using oil pressure)
[krW, krO, krG] = f.relPerm(sW);
bW   = f.bW(p);
rhoW = bW .* f.rhoWS;
rhoWf= S.faceAvg(rhoW);
mobW = f.tranMultR(p) .* krW ./ f.muW(p);

% upstream weighting
gdz  = S.Grad(G.cells.centroids)*grav';
dpW  = S.Grad(p - f.pcOW(sW)) - rhoWf.*gdz;
upc  = (double(dpW) <= 0);
bWvW = -S.faceUpstr(upc, bW.*mobW) .* S.T .* dpW;

% discrete equation
eq{2} = S.Div(bWvW) + (S.pv/dt) ...
       .*(f.pvMultR(p).*bW.*sW - f.pvMult(p0).*f.bW(p0).*sW0 );
```

Fig. 6—Discrete operators and discretization of the water equation for a black-oil model that uses oil pressure, water and gas saturations as primary unknowns. For brevity, the implementation of well equations is not included and certain parts of the code has been slightly modified for pedagogical purposes.

Discrete flow equations. For brevity, we only discretize the equation for the water phase; the oil and gas equations follow the same pattern, except for the obvious modifications. Using the abstract operators introduced above and dropping subscripts for simplicity, the discrete water equations read,

$$\begin{aligned}\frac{1}{\Delta t}\Big(\boldsymbol{\phi}(\boldsymbol{p}[c])\,\boldsymbol{b}(\boldsymbol{p}[c])\,\boldsymbol{s}[c]\Big)^{n+1} - \frac{1}{\Delta t}\Big(\boldsymbol{\phi}(\boldsymbol{p}[c])\,\boldsymbol{b}(\boldsymbol{p}[c])\,\boldsymbol{s}[c]\Big)^{n} + \mathtt{div}(\boldsymbol{v})[c]^{n+1} - (\boldsymbol{bq})[c]\big)^{n+1} = 0,\\ \boldsymbol{v}[f] = -\mathtt{upw}(\boldsymbol{\lambda})[f]\boldsymbol{T}[f]\Big(\mathtt{grad}(\boldsymbol{p}-\boldsymbol{p}_c^{ow})[f] - g\,\mathtt{avg}(\boldsymbol{\rho})[f]\,\mathtt{grad}(\boldsymbol{z})[f]\Big).\end{aligned} \tag{12}$$

Here, $\lambda = bk_r/\mu$ and $p_c^{ow} = p_w - p_o$ is the capillary pressure between the water and the oil phase, whereas $\mathtt{avg}(\boldsymbol{p})[f]$ denotes a face-valued pressure computed as the arithmetic average of the pressures in the two adjacent cells. Finally, $\mathtt{upw}$ is the phase-upwind function,

$$\mathtt{upw}(\boldsymbol{\lambda})[f] = \begin{cases}\boldsymbol{\lambda}[N_1(f)], & \text{if } \mathtt{grad}(\boldsymbol{p}-\boldsymbol{p}_c^{ow})[f] - g\,\mathtt{avg}(\boldsymbol{\rho})[f]\mathtt{grad}(\boldsymbol{z})[f] > 0,\\ \boldsymbol{\lambda}[N_2(f)], & \text{otherwise.}\end{cases} \tag{13}$$

Fig. 6 outlines how this discretization is implemented using `MRST-AD`. Here, the single-phase transmissibility and the definition of operators for discrete derivatives, face and cell averages, and upstream weighting have all been collected in a special structure `S` along with a few other quantities like the pore volume at reference pressure. Likewise, all functions that are used to evaluate fluid properties like the phase densities, viscosities, formation-volume factors, and relative permeabilities have been collected in an object `f`. This object is usually called the *fluid object*, but for convenience it may also contains other constitutive relationships like the pressure-dependent transmissibility multiplier `f.tranMultR` and the multiplier `f.pvMultR` used to account for pressure variations in the pore volume `S.pv`. MRST has several fluid models that are either hand-coded, based on analytic expressions, or auto-generated from industry-standard input decks. These objects can be passed on to the black-oil code outlined above and modified and extended if necessary without changing the discretized equations. However, if you change the functional dependencies so that properties depend on new primary variables, these changes must obviously also be implemented in the discrete equations in Fig. 6, e.g., as shown in the previous section. In either cases, any changes in the linearized equations will be accounted for automatically by the AD-class which ensures that the correct Jacobian is computed as long as all constitutive relationships are expressed by arithmetic operations. If not, you can still make your constitutive relationships AD-compatible by making sure that they output the correct derivatives when called with AD-variables.

Nonlinear and linear solvers. In the single-phase simulator shown in Fig. 3, the nonlinear solver and the loop used to evolved the solution in time were very simple. Industry-standard black-oil models require a much more complicated setup. First of all, the input deck will in most cases describe a simulation schedule that be considered to consist of a sequence of well controls that each are active for a certain time period. The simulator may sometimes be able to compute a control period in a single time step, but in most cases the period needs to be subdivided into several local steps to ensure a stable and reasonably accurate solution. This is typically done by reducing the time step if the Newton solver does not converge within a prescribed number of iterations. The Newton step may also need some kind of regularization; we use a dampening factor when repeated steps do not decrease the residuals. For small systems (a few thousand cells), the linearized system inside the Newton loop can be efficiently solved

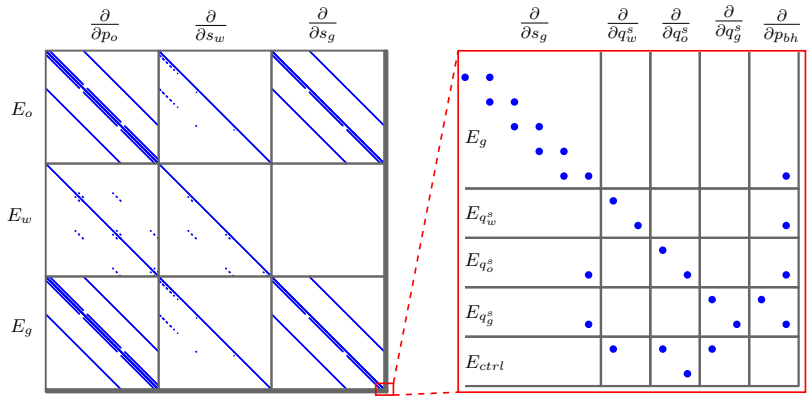

Fig. 7—Illustration of the structure of the linearized black-oil equations and all the different sub-Jacobians that make up the overall linear system for a problem with two wells.

using direct sparse methods, while for larger systems an iterative approach is appropriate. In MRST, the current option is a CPR-preconditioned GMRES method. Our block representation of Jacobians is well suited for efficient and compact implementation of CPR-type preconditioners and for elimination of variables. To illustrate, in **Fig. 7** we have plotted the sparsity pattern of the Jacobian for a black-oil system containing two wells. In all, there are seven systems of equations; the three reservoir equations (Eq. 11), three equations setting the well surface rates to the sum of the perforation contributions, and finally, the control equation ensuring that each well operates under the prescribed control (bottom-hole pressure, surface rate, etc). Similarly, there are seven primary vector-variables: reservoir oil pressure, water saturation, gas saturation or gas-oil ratio (depending on the state of the grid-cell), surface rates for each of the three components, and finally the bottom-hole pressures in the wells. Accordingly, for example the block at position $(3, 2)$ is the partial derivative of the discretized gas-equation with respect to the water saturation variables, i.e., $\partial \boldsymbol{E}_g / \partial \boldsymbol{s}_w$. When MRST encounters a linear system of this form, the solution procedure proceeds as follows:

1. Eliminate the well rate and bottom-hole pressure variables resulting in a system $J\boldsymbol{x} = \boldsymbol{b}$, where J consists of 3×3 blocks $J(m, n)$, $m, n = 1, 2, 3$, and $\boldsymbol{x}$ consists of the pressure and saturation variables. For standard well models only inversion of diagonal matrices are needed in this process.

2. Set the first block-row in the system equal to the sum of the three block-rows, i.e., $J(1, m) = \sum_n J(n, m)$. The purpose of this is that $J(1, 1)$ should resemble a pressure equation and become close to elliptic. In summing the equations we leave out rows that may harm the desired diagonal dominance in $J(1, 1)$. The logic used for this is adopted from Gries et al. (2014).

3. Set up the two-stage preconditioner $M_2^{-1} M_1^{-1}$:

   (a) The first preconditioner M_1^{-1} is set up to solve the near elliptic subsystem $J(1, 1)\delta \boldsymbol{p}^{i+1} = -\boldsymbol{r}_p^i$ to obtain the pressure update $\delta \boldsymbol{p}^{i+1}$. For large systems an algebraic multigrid solver is preferable.

   (b) The second preconditioner M_2^{-1} is based on an incomplete LU-decomposition $LU \approx J$ of the full system, and is set up to perform a variable update $\delta \boldsymbol{x}^{i+1}$ on the full set of variables by solving $LU\delta \boldsymbol{x}^{i+1} = -\boldsymbol{r}_x^i$.

4. Solve the full system with GMRES using $M_2^{-1} M_1^{-1}$ as preconditioner.

5. Recover rate and bottom-hole pressure variables.

When using AMG for the pressure system, it is vital that this system indeed is close to elliptic. We have explored several CPR-versions, but by far the most robust and simple in our experience is the one proposed by Gries et al. (2014). In addition, the above approach is straight forward to generalize when additional variables (e.g., temperature, polymer) and equations are added to the system.

Object-oriented AD simulators. Originally, the AD simulators in MRST were written explicitly for the purpose of production optimization using field models with black-oil fluid properties (Raynaud et al. 2014). These were over time extended to a wide variety of other physical models including EOR options like polymer and surfactant (Jørgensen 2013; Hilden et al. 2014), geochemical effects, thermal effects, and vertical equilibrium models for geological CO_2 storage (Nilsen et al. 2014b,a). In the process, however, the code became unwieldy to work with and included a wide variety of options that only were applicable to certain physical models. On the other hand, we also observed that a lot functionality is quite generic and can be reused from one simulator to another with no or few modifications. We therefore decided to separate the implementation of physical models,

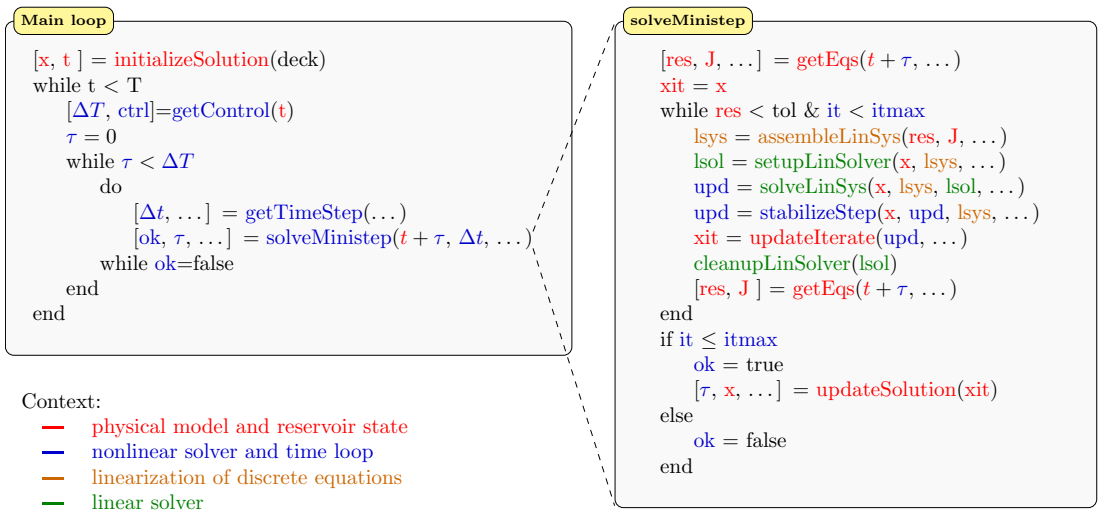

Fig. 8—The time-loop of a fully-implicit simulator organized into specific numerical contexts: physical model and updates to reservoir state, nonlinear iteration, linearization of model equations, linear solver, etc.

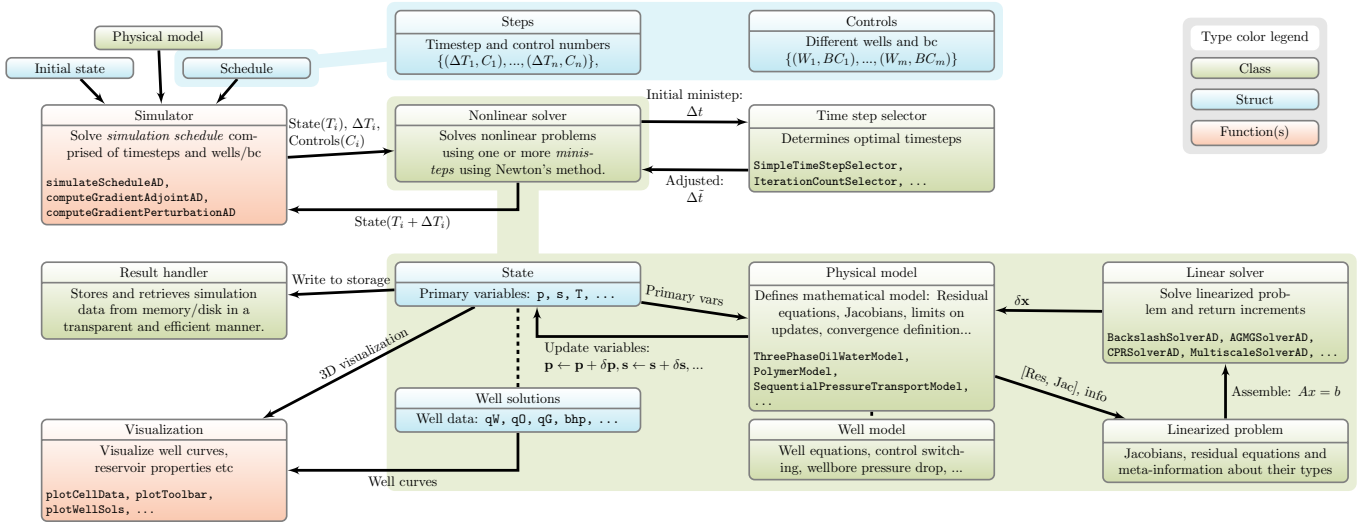

Fig. 9—The different components that make up an AD simulator colorized by the type of the corresponding construct (class, struct, or function). Notice, in particular, how the nonlinear solver uses multiple components to solve each mini-step on behalf of the simulator function.

discrete operators, discretizations, nonlinear solver and time-stepping, and assembly and solution of the linear system, and only expose the details that are needed within each of these contexts. As a result, the entire framework was rewritten from the ground up using an object oriented model. During the design phase, we emphasized separating the nonlinear solution process from the underlying physical models so that the advances in solver technology can immediately be put to use for novel physical models.

Fig. 8 summarizes how the time-loop of a fully-implicit simulator can be organized into numerical contexts that focus on specific parts of the overall algorithm, whereas **Fig. 9** gives more details how these are realized using components that represent different constructs. For the AD-framework, there is a distinction between the main classes and the helper classes. The main classes are the nonlinear solver class that implements the Newton solver, the physical model class that implements the discrete model equations and rules for how to updated the physical state based on increments computed by the nonlinear solver, and the linear solver class that solves the linearized discrete problems. The other classes serve to enhance the main classes in different ways. For instance, by itself the nonlinear solver only cuts time steps when the nonlinear iteration count exceeds some threshold, but if it is enhanced with a time-step controller, the time steps can also be dynamically adjusted to obtain the desired accuracy and solution speed. In much the same way the reservoir-model class can use a well-model class to assemble complex well-control equations for multi-segmented wells with for example both bottom-hole pressure and liquid-rate controls.

The advantage of this level of separation between components is that it enables researchers to focus on the part they are

Fig. 10—The implementation of a simple polymer model by modifying the existing oil/water model, whose discrete operators and flow equations for the water phase were outlined in Fig. 6.

interested in. When prototyping a novel physical model, there is in many cases no need to reimplement features of the nonlinear solver just because the model equations change. Likewise, anyone interested in prototyping advanced linear solvers can easily test them on a wide variety of problems without any knowledge of how these models are implemented. To illustrate this process of rapid prototyping, we will demonstrate how an existing two-phase, compressible model can be extended with a simple polymer model.

A simple polymer model. The polymer model considered here is simplified from the implementation in MRST for pedagogical reasons. The purpose here is to show a process in which an existing model is extended by an additional component conservation equation that alters the properties of the water phase and introduces a hysteretic behavior. The polymer component will exist in the water phase and is used to make the water more viscous and thus less mobile. Polymer diluted in water usually has so low concentration that viscosity is the only water property affected. Here, we will use a standard mixing model (Todd and Longstaff 1972),

$$\begin{aligned} \mu_{p,\text{eff}} &= \mu_m(c)^\omega \mu_p^{1-\omega}, & \mu_p &= \mu_m(c_m)) \\ \mu_{w,\text{eff}} &= \Big[\frac{1-\bar{c}}{\mu_{w,e}} + \frac{\bar{c}}{\mu_{p,\text{eff}}}\Big]^{-1}, & \mu_{w,e} &= \mu_m(c)^\omega \mu_w^{1-\omega}, \quad \bar{c} = c/c_m, \end{aligned} \tag{14}$$

where c is the polymer concentration, c_m is its maximal attainable value, and $\omega \in [0,1]$ is a mixing parameter. By introducing the multiplier $m(c) = \mu_m(c)/\mu_w$, we can write the second equation as $\mu_{w,\text{eff}} = \mu_w m(c)^\omega/[1-\bar{c}+\bar{c}/m(c_m)]$. In our model, we also account for polymer being adsorbed onto the rock, which is assumed to take place instantaneously so that the amount of polymer adsorbed C_p^a is a function of c and the maximal attained value $c_{\max}$. The adsorbed polymer will fill up and block pores and hence reduce the effective permeability. The reduction in permeability will be modelled as a *hysteresis effect*, making the permeability at a point a function of the largest polymer value seen at this point. That is, we introduce the reduction factor $R_k(c, c_{\max}) = 1 + (\gamma - 1)C_p^a(c, c_{\max})/C_{\max}^a$, where γ is the ratio of the initial water mobility to the water solution mobility after polymer flooding and $C_{\max}^a$ is the maximum possible absorbed polymer. We can then write out the conservation equations for the water phase and the polymer components:

$$\begin{aligned} \partial_t(\phi b_w s_w) + \nabla\cdot(b_w\vec{v}_w) - b_w q_w = 0, \qquad & \vec{v}_w = -\frac{k_{rw}\mathbf{K}}{\mu_{w,\text{eff}} R_k(c,c_{\max})}(\nabla p_w - \rho_w g \nabla z), \\ \partial_t\big[\phi b_w s_w c + (1-\phi_{\text{ref}})C_p^a(c,c_{\max})\big] + \nabla\cdot(b_w\vec{v}_{wp}c) - b_w q_w c = 0, \qquad & \vec{v}_{wp} = -\frac{k_{rw}\mathbf{K}}{\mu_{p,\text{eff}} R_k(c,c_{\max})}(\nabla p_w - \rho_w g \nabla z). \end{aligned} \tag{15}$$

To show how the implementation will appear in the object-oriented framework, we refer to **Fig. 10**. First of all, we let the polymer model inherit the properties and functions of the oil water model it is based on and extend the set of variables to include c and $c_{\max}$ in the function `getVariableField`. In function `updateState` we impose that $c \in [0, c_m]$, whereas the function `updateAfterConvergence` updates the maximal attained value, $c_{\max}(x,t) = \max_{\tau<t} c(x,\tau)$, needed for the hysteretic

modeling in $R_k(c, c_{\max})$. The discrete equations are programmed by copying the corresponding code for the oil–water system and introducing the modifications outlined in red in the box to the right in Fig. 10. Finally, one the physical properties of polymer and the polymer-water mixture must be implemented in the fluid object. In our current implementation, this is done by parsing industry-standard input decks, but could equally well have been hand-coded analytical formulas. The parsing is automated in the sense that for each new keyword you want to interpret, you create a file called `assign⟨KEYWORD⟩.m` in the `props` directory of the `ad-fi` module. In our case, this amounted to less than fifty extra lines of code to interpret the polymer keywords and set up functions that interpolate the tabulated values in the input deck correctly. Apart from the changes outlined above, everything else works automatically, including time-step control, (CPR-type) preconditioners, adjoint methods for computing gradients and sensitivities, etc.

Numerical Experiments

An important aspect when presenting a new model or computational method is to compare it with existing models and methods and possibly explain any differences. To support this, we have got to lengths to ensure that `MRST-AD` is able to reproduce the results of a leading commercial simulator on standard benchmarks as well as on real-field models. In the following, we present the results of three such validation studies. In addition, we also briefly illustrate use of the thermal simulator obtained by rapid prototyping.

SPE 1. The first project comparing black-oil reservoir simulators was organized by Odeh (1981) and describes 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, while 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 dissolved gas-oil ratio (R_s) throughout the model. The original problem was posed to study ten years of production; herein, we only report results for the first 1216 days. **Fig. 11** compares the solutions computed by `MRST-AD` and Eclipse 100 using the same time steps. The solutions are qualitatively similar, and after a careful investigation, we discovered that the minor discrepancies can be explained by subtle differences in how the two simulators interpolate tabulated fluid data. Whereas Eclipse 100 interpolates $1/(\mu_o B_o)$ as a product, our software first interpolates μ_o and B_o and then computes the product. By changing the interpolation, we obtained identical results.

SPE 9. The Ninth SPE Comparative Solution Project (Killough 1995) was introduced twenty years ago to compare contemporary black-oil simulators and investigate "complications brought about by heterogeneity in a geostatistically-based permeability field". The reservoir is described by a $24 \times 25 \times 15$ grid, having a 10 degree dipping-angle in the x-direction. By current standards, the model is quite small, but contains a few features that will still pose challenges for black-oil simulators. The well pattern consists of twenty-five producers and one water injector: the producers initially operate at a maximum rate of 1500 STBO/D, which is lowered to 100 STBO/D from day 300 to 360, and the raised up again to its initial value until the end of simulation at 900 days. The water injector was set to a maximum rate of 5000 STBW/D with a maximum bottom-hole pressure of 4000 psi at reference depth. This setup will cause free gas to form after approximately one hundred days as the reservoir pressure is reduced below the original saturation pressure and migrate to the top of the reservoir. During the simulation most of the wells convert from rate control to pressure control. A second problem is a discontinuity in the water-oil capillary pressure curve, which may cause difficulties in the Newton solver when saturations are changing significantly. **Fig. 12** compares production curves computed by our simulator and a commercial simulator using the same time steps. For MRST, we have also included the mini-steps that are taken within the outer nonlinear loop. As is evident from the plots, there is good match between our simulator and the commercial code.

Voador field model. The Voador field is located in the Campos Basin, approximately one hundred miles off the coast of Brazil. It is a solution-gas-drive field in which the pressure has been depleted to below the bubble point, see (Hasan et al. 2013). The field consists of two non-communicating reservoirs. The south wing started to produce in November 1992 and a multilateral injector was drilled in 1999 to maintain reservoir pressure. The oil viscosity in the reservoir is particularly high with an oil-to-water ratio varying from 8 (at 200 bar) to 40 (at 500 bar) for saturated oil. Initially, there was no free gas, and the oil and water phases were fully separated by gravity (capillary effects are neglected). After peak production passed, additional wells were drilled sequentially. As the field entered its mature stage, the total oil production became less than 800 Sm^3/day and the water cut exceeded 90% in some producers. The reservoir has also produced free gas after the pressure depleted below the bubble point. In 2011, the gas-oil ratio for every production well was approximately 80 $\mathrm{Sm}^3/\mathrm{Sm}^3$.

We have at our disposal a schedule based on historical data spanning a period of nineteen years, from 1992 to 2011, and the corresponding output from a commercial simulator. Altogether, the south wing is equipped with eight wells: one injector and seven producers. In the schedule, the injector is controlled by water rate, while the producers are controlled by oil rate. A simplified two-phase version of this simulation model was used by Hasan et al. (2013) to study the combination of long-term and short-term production optimization. Likewise, a shorter period of the three-phase, black-oil model was used by Raynaud et al. (2014) to study upscaling and adjoint methods for production optimization. Herein, we will study the full history of the south wing, for which the full model consists of 86 389 cells and constitutes five disconnected parts. We pick the largest one; after

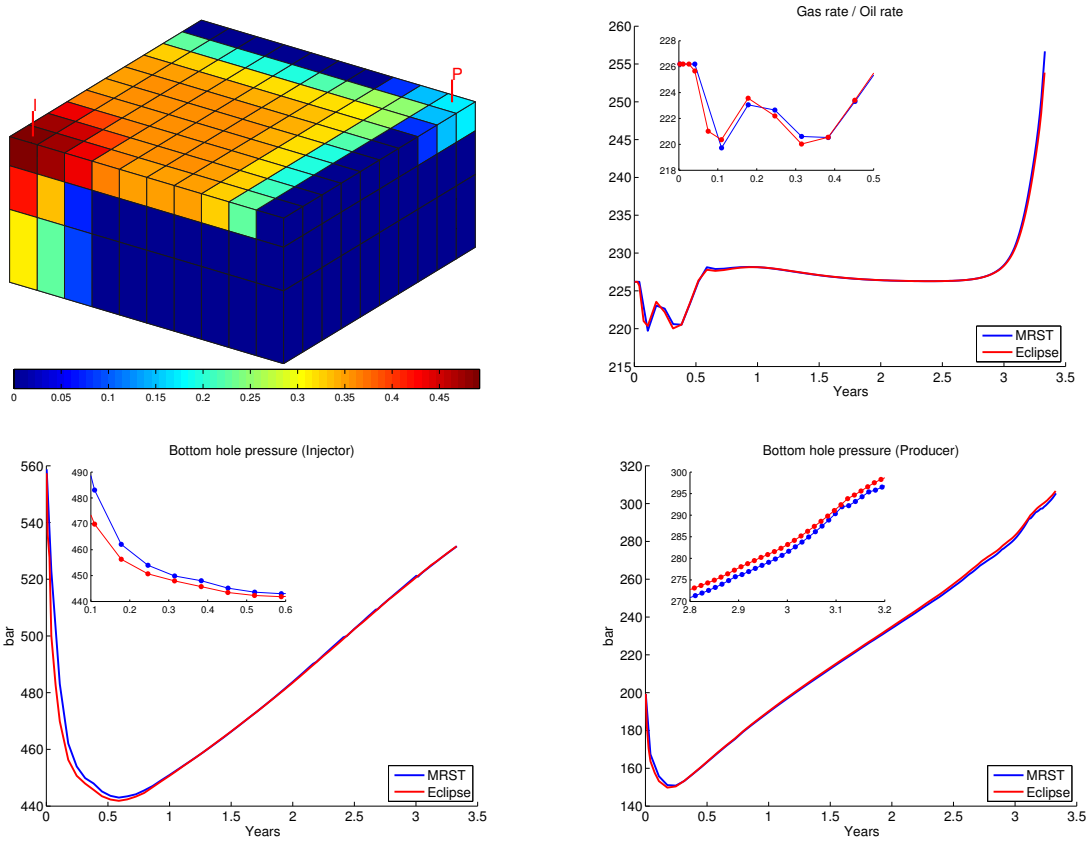

Fig. 11—Solutions computed by MRST-AD and Eclipse 100 for the SPE 1 benchmark case. The upper-left plot shows the gas saturation after 1216 days computed by our software. The other plots compare production curves. The minor discrepancies are caused by subtle differences in how tabulated fluid data are interpolated.

removal of cells with zero porosity, the model contains 26 911 active cells. The water-oil contact is specified at different depths in two connected regions so that the initial state is not at equilibrium. The multilateral injector penetrates five disconnected sets of cells, and following the commercial simulator, these cells were ordered based on centroid depths.

In **Fig. 13**, the controls corresponding to the historical schedule are shown. We recognize the periods of primary recovery where only two producers are used and of secondary recovery after the injector has been started. In **Fig. 14**, we compare the results obtained by our and a commercial simulator for the bottom-hole pressures and the water rates at the wells. There is a globally a very good match between the two sets of results. Most of our efforts in producing these results have been concentrated on the treatment of the wells, understanding the logic the commercial simulator follows and reproduce it in code. The first producer, for which the differences in the water rate are the highest, has a complicated schedule. We believe that the discrepancies we observe are caused by the fact that we were not able to fully reproduced the choices the commercial simulator makes in its treatment of wells.

Thermal simulation. To demonstrate the thermal code from Figs. 3 – 5 we consider a simple $200 \times 200 \times 50$ m box-shaped reservoir realized on a $10 \times 10 \times 10$ Cartesian grid. The reservoir has homogeneous permeability of 30 mD, constant porosity 0.3 at reference pressure 200 bar, and a rock compressibility of 10^{-6} bar^{-1}. No-flow conditions are assumed along all boundaries. For the fluid model, we use

$$\rho(p,T) = \rho_r\big[1+\beta_T(p-p_r)\big]e^{-\alpha(T-T_r)}, \qquad \mu(p,T) = \mu_0\big[1+c_\mu(p-p_r)\big]e^{-c_T(T-T_r)}, \tag{16}$$

where $\rho_r = 850$ kg/m^3 is the density and $\mu_0 = 5$ cP the viscosity of the fluid at reference pressure $p_r = 200$ bar and temperature $T_r = 300$ K. The constants are $\beta_T = 10^{-3}$ bar^{-1}, $\alpha = 5\times10^{-3}$ K^{-1}, $c_\mu = 2\times10^{-3}$ bar^{-1}, $c_T = 10^{-3}$ K^{-1} and $c_p = 4\times10^3$ J/kg. The reservoir is initially at hydrostatic equilibrium and has constant temperature of 300 K. We use a simple linear relation for the enthalpy, which is based on the thermodynamical relations that give

$$\mathrm{d}H = c_p\,\mathrm{d}T + \left(\frac{1-\alpha T}{\rho}\right)\mathrm{d}p, \qquad \alpha = -\frac{1}{\rho}\frac{\partial\rho}{\partial T}\Big|_p. \tag{17}$$

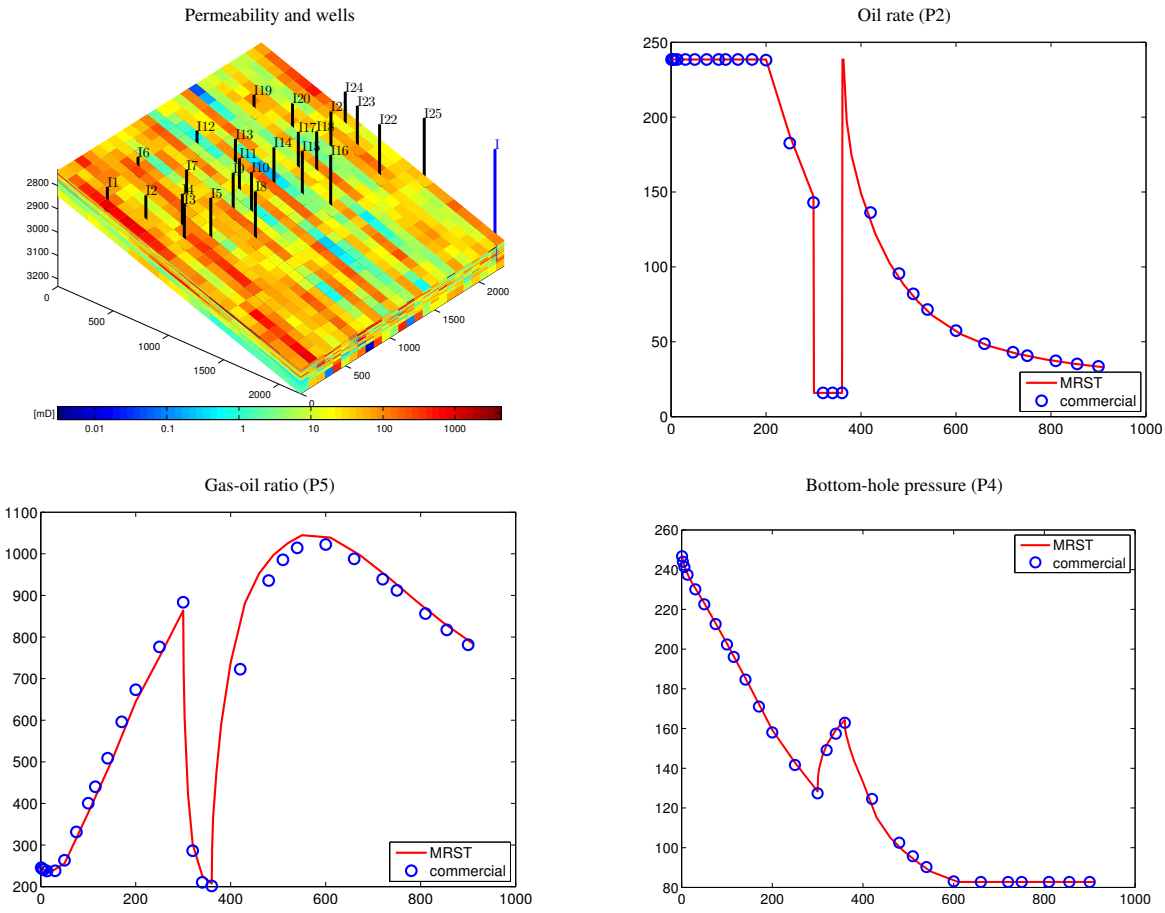

Fig. 12—Solutions computed by MRST-AD and Eclipse 100 for the SPE 9 benchmark case.

The reservoir is drained from a single horizontal well that operates at a constant bottom-hole pressure of 100 bar and is perforated in cells with indices $i = 2$, $j = 2, \ldots, 9$, and $k = 5$. **Fig. 15** shows snapshots of the pressure and temperature evolution. The open well will create a pressure draw-down that gradually propagates into the reservoir and as more fluid is drawn from the reservoir, the pressure decays towards a steady state with pressures in the interval [101.2,104.7] bar. The expansion and flow of fluid will cause an instant cooling near the well-bore that gradually propagates into the reservoir and diffuses out towards a temperature of approximately 295.5 K.

The change in temperature of an expanding fluid will not only depend on the initial and final pressure, but also on the manner in which the expansion is carried out. In a free expansion, the internal energy is preserved and the fluid does no work. 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 expansion reads

$$\mathrm{d}E + \frac{p}{\rho V}\,\mathrm{d}V = \mathrm{d}E + p\,\mathrm{d}(\frac{1}{\rho}) = 0. \tag{18}$$

In a Joule–Thompson process, the enthalpy remains constant while the fluid flows from higher to lower pressure under steady-state conditions and without change in kinetic energy. Our case is a combination of these three processes, and to demonstrate their interplay, we study six different cases in which we vary the value of α while keeping the compressibility β_T constant. Some of the values we choose have little physical relevance, but serve to illustrate different effects and the power of our framework. **Fig. 16** shows the minimum, average, and maximum temperature in the reservoir as a function of time for six different values of α. We have also included horizontal lines that indicate the change in temperature that would result from the total pressure drop that takes place if the adiabatic expansion was governed by one and only one of the three processes.

The change in behavior between the two first and the other plots is associated with the change of sign of $\partial E/\partial p$,

$$\mathrm{d}E = \left(c_p - \frac{\alpha p}{\rho}\right)\mathrm{d}T + \left(\frac{\beta_T p - \alpha T}{\rho}\right)\mathrm{d}p, \qquad \beta_T = \frac{1}{\rho}\frac{\partial \rho}{\partial p}\Big|_T. \tag{19}$$

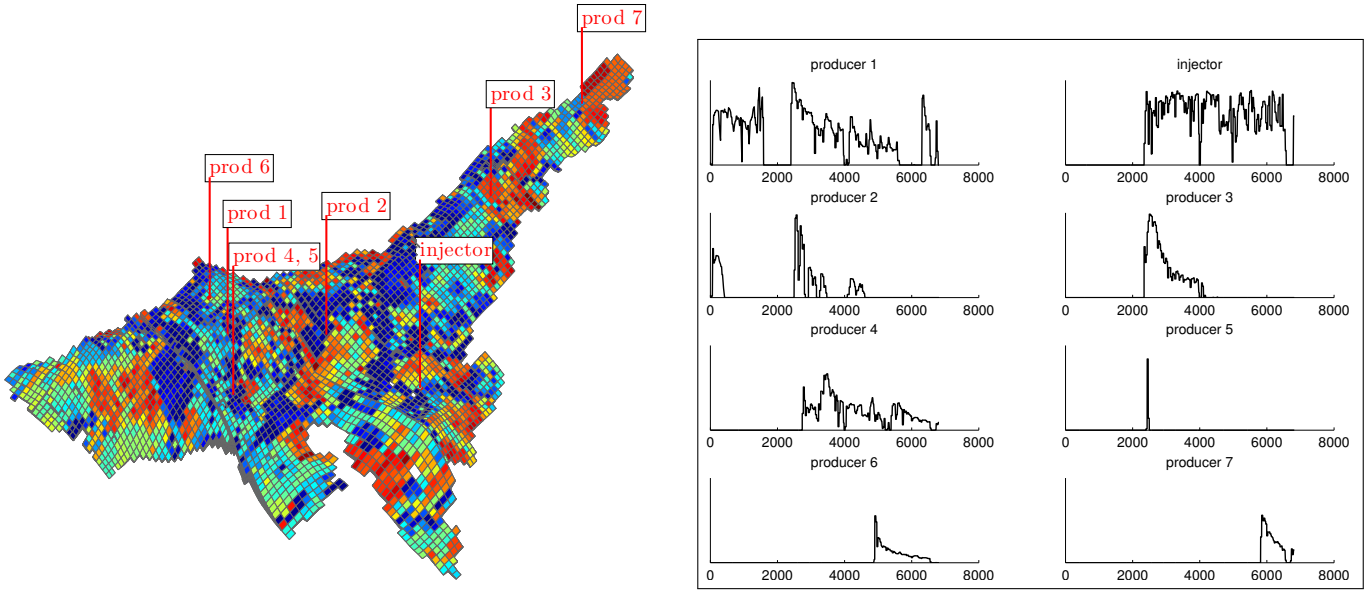

Fig. 13—Representation of the south wing of the Voador reservoir (left) with the well locations and the permeability field. Control variables for the historical schedule (right): Water injection rate for the injection well and oil production rates for the production wells. Zero pressure values correspond to times where the well has been shut down.

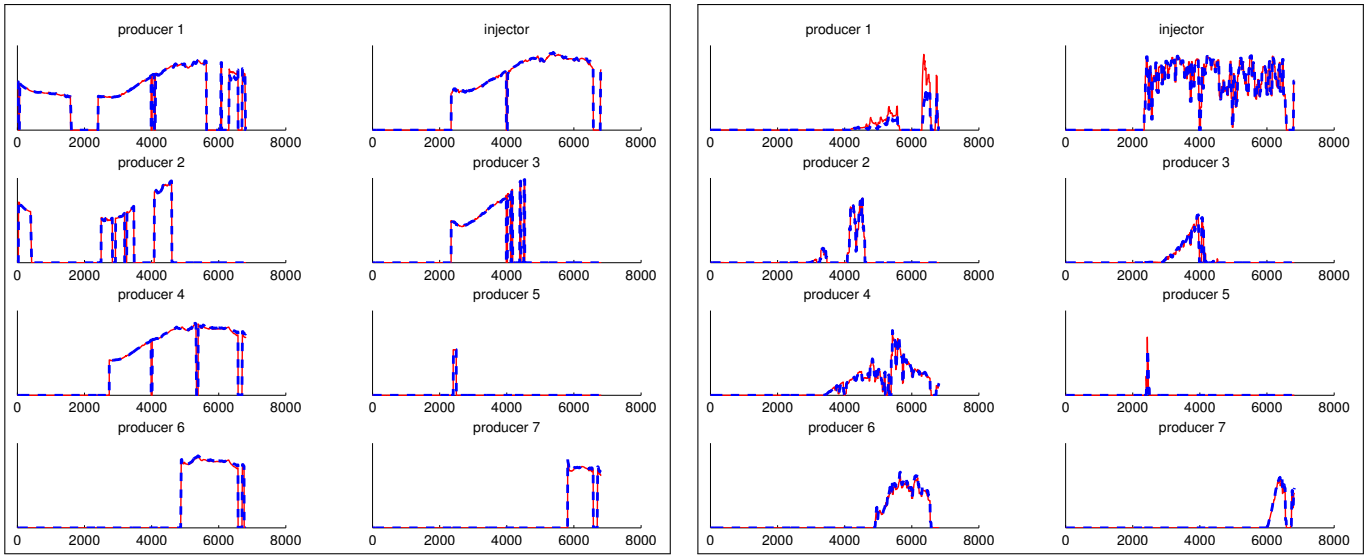

Fig. 14—Bottom-hole pressures (left) and water rates (right) at the wells. The results of MRST-AD and Eclipse 100 are in red and blue, respectively.

For the two first plots $\alpha T < \beta_T p$ 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 these two plots. The Joule–Thomson coefficient $(\alpha T - 1)/(c_p \rho)$ is negative for plot number one to four, 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 these cells go from having the lowest to having the highest temperature in the reservoir because of heating from the moving fluids. When we go to systems with positive Joule–Thomson coefficient in the two last plots, we notice that the minimum temperature is observed at the well for a longer time. Another interesting feature of the system is the kink in the minimum temperature curve, which appears when the point of minimum temperature moves from being at the bottom of the front side to the far back of the model. 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.

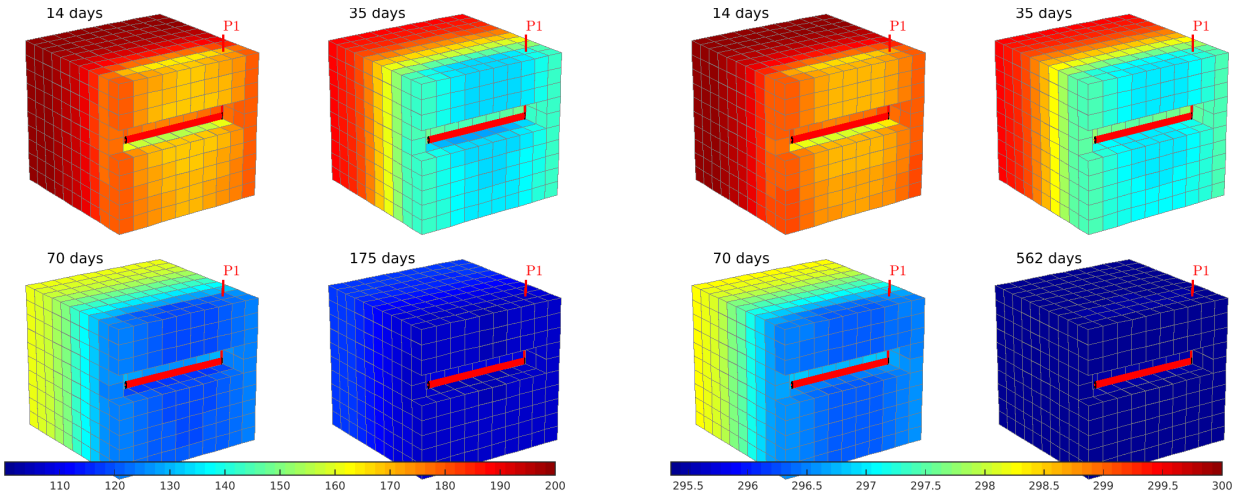

Fig. 15—Evolution of the pressure (left) and temperature (right) computed by the single-phase, thermal simulator. For clarity, the vertical dimension is scaled by a factor four.

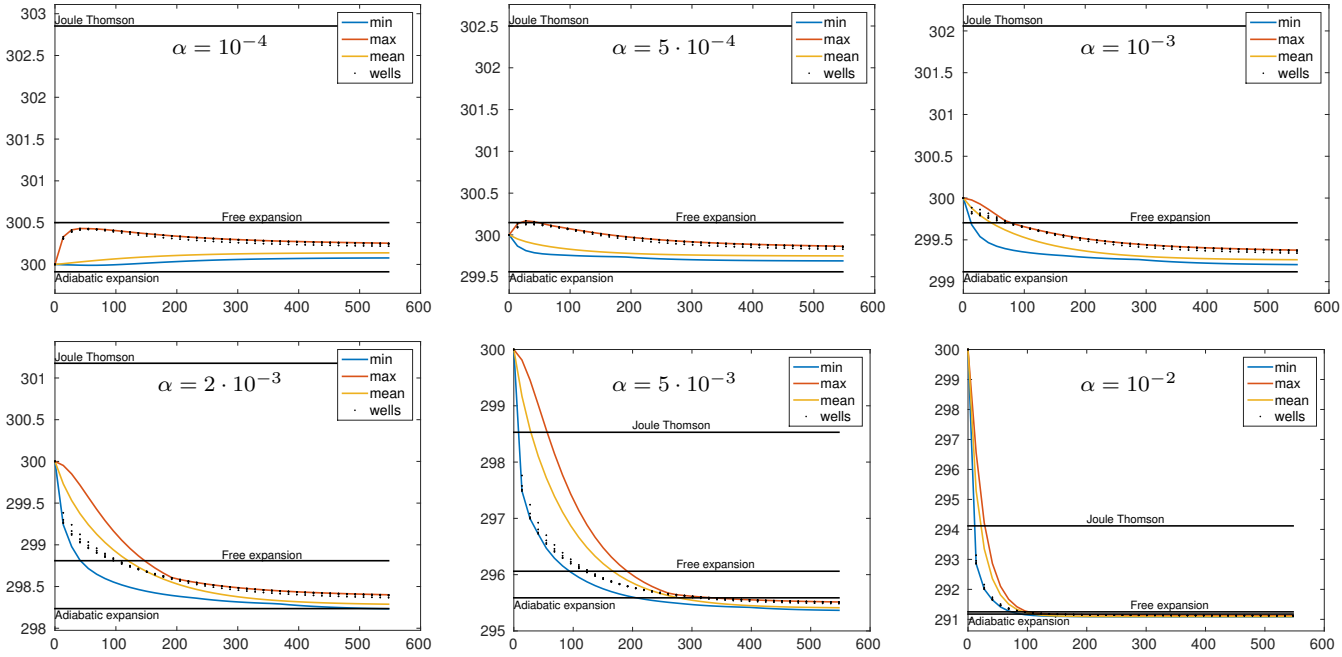

Fig. 16—Evolution of the minimum, maximum, and average temperature as well as the temperature in each of the well perforations for different values of the thermal expansion factor α [K^{-1}], all plotted as functions of time in unit days.

Computing linearized responses for the thermodynamical functions is particularly simple using automatic differentiation. As an example, **Fig. 17** shows how to linearize the primary enthalpy function and use the result to solve for the temperature resulting after a Joule–Thomson expansion. Similarly, we can compute the temperature after a reversible expansion, which is not 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 Eq. 18. 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.

Polymer injection. Earlier in the paper on page 12 we discussed a simplified two-phase polymer model and showed in Fig. 10 how this could easily be implemented as a simple extension of a two-phase black-oil model. Here, we will consider a more complex three-phase model that has oil, water, and wet gas. Fluid parameters for the three-phase model are taken from a public model of the Norne field in the Norwegian Sea, see (Norne 2012; Package 2), whereas the polymer data are representative of a

Linearizing enthalpy

```
[p,T] = initVariablesADI(pref,Tref);
dp   = p_final-pref;
hf   = Hf(p,T);
dHdp = hf.jac{1};
dHdT = hf.jac{2};
Tjt  = Tref - dHdp*dp/dHdT; % Joule Thomson temperature
```

Linearizing adiabatic expansion

```
[p,T] = initVariablesADI(pref,Tref);
dp   = p_final - pref;
hf   = Ef(p,T) + double(p)./rho(p,T);
dHdp = hf.jac{1};
dHdT = hf.jac{2};
Tjt  = Tref - dHdp*dp/dHdT; % Linearized adiabatic temperature
```

Fig. 17—Two examples of the computation of linearized responses of thermodynamical functions. The left box shows how to calculate the proper linear differential of enthalpy density for a Joule–Thompson process, while the right box shows how to calculate the incomplete differential in Eq. 19 associated with a reversible expansion process.

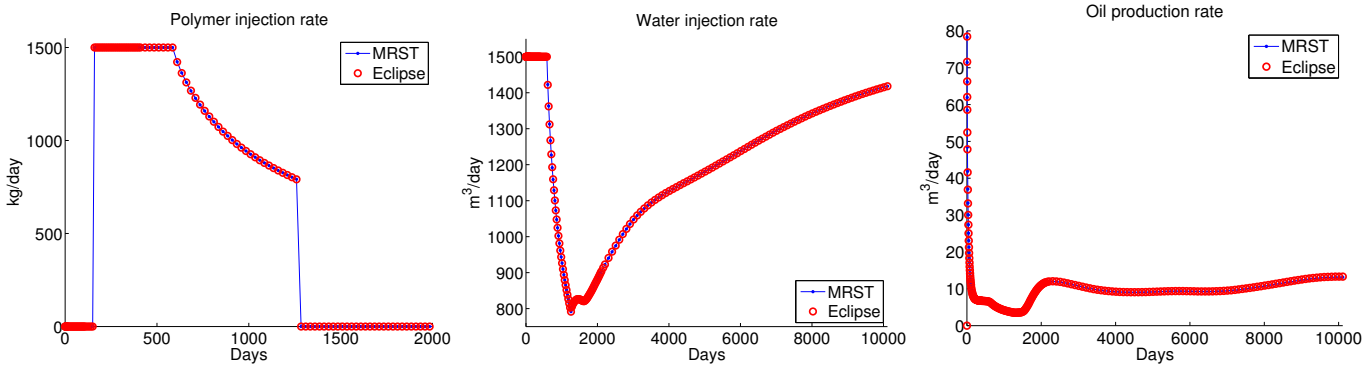

Fig. 18—Water curves predicted by MRST-AD and Eclipse 100 for the simulation of a three-phase polymer injection scenario. For clarity, the figure only shows every third point in the solution computed by the commercial simulator.

polymer injection scenario in another field. The reservoir geometry, however, is very simple and consists of a $4000 \times 200 \times 75$ m^3 box represented on a regular $20 \times 1 \times 3$ Cartesian grid, with an injection well completed in the cells to the far west of the bottom two layers and a production well completed in the upper layer to the far east. The petrophysical properties are homogeneous with 0.1 porosity and [0.21,0.21,0.022] mD permeability. The bottom layer initially contains water, the middle layer contains a mixture of 57.4% water and 42.6% water, and the upper layer a mixture of 93.4% gas and 6.6% oil. The reservoir is produced by first injecting water for 150 days, then a polymer slug for 1110 days, followed by 8850 days of water injection. The injector has a target rate of 1500 m^3/day and is limited by a bottom-hole pressure of 450 bar, whereas the producer operates at a fixed pressure of 200 bar. **Fig. 18** shows a comparison between solutions computed by `MRST-AD` and Eclipse 100. What is remarkable about this example is neither the case, which is very simple in terms of reservoir description, nor the match with the commercial solver, but the fact that the implementation and match was obtained in less than one week by someone who was not very familiar with the software.

Concluding Remarks

The software `MRST-AD` presented herein is the result of many years of research on computational methods for reservoir simulation and CO_2 sequestration. Our main motivation for developing and maintaining it has been to simplify our own research and make the members of our research team more productive. For this reason, the software has been developed using a vectorized scripting language, which in our experience has a lower user threshold and a much shorter development cycle than any compiled language we so far have worked with. On one hand, the software offers a relatively rich grid structure to provide generality and flexibility when developing new discretizations, and on the other hand, it provides a set of discrete differential operators combined with automatic differentiation to increase productivity when developing new mathematical models, time-stepping methods, and nonlinear solvers. To simplify the process of verifying and validating new models and methods on problems of real-life complexity, the software also offers functionality to read and process industry-standard input decks. Functionality has also been developed so that the software can be used as a black-box simulator that offers a reasonable subset of the features seen in commercial simulators. Obviously, simulators developed in `MRST-AD` will not be as fast as a commercial simulator, but in many cases a higher runtime (typically three to ten times) is compensated by the general ability to modify, replace, and extend any part of the simulator. By releasing the software as open source, and by writing this paper, we hope that other researchers can benefit from our work and be more productive. Use of open source is also an important pillar for the higher goal of reproducible science. The current public release is mainly focused on fully-implicit, black-oil type simulators, but we also have prototype versions of new modules supporting more advanced modeling options such as geochemistry, thermal and geomechanical effects that are scheduled for release in our biannual schedule.

Acknowledgments

The authors wish to thank Prof. Jon Kleppe at NTNU, Trondheim for sharing input decks for the SPE 1 and SPE 9 benchmark cases and Alex Teixeira from Petrobras for providing and sharing the simulation model of the Voador field. We also thank Jostein R. Natvig, who over the past years has contributed a lot to the development of MRST, and our colleagues Sindre T. Hilden, Atgeirr F. Rasmussen, Odd Andersen, and Kai Bao for helpful discussions. In particular, Kai Bao, developed the code and data set for the polymer injection example.

References

Aarnes, J. E., Gimse, T., and Lie, K.-A. 2007a. An introduction to the numerics of flow in porous media using Matlab. In Hasle, G., Lie, K.-A., and Quak, E., editors, *Geometrical Modeling, Numerical Simulation and Optimisation: Industrial Mathematics at SINTEF*, pages 265–306. Springer Verlag, Berlin Heidelberg New York. doi: 10.1007/978-3-540-68783-2_9.

Aarnes, J. E., Hauge, V. L., and Efendiev, Y. 2007b. Coarsening of three-dimensional structured and unstructured grids for subsurface flow. *Adv. Water Resour.*, 30(11):2177–2193. doi: 10.1016/j.advwatres.2007.04.007.

Aarnes, J. E., Krogstad, S., and Lie, K.-A. 2006. A hierarchical multiscale method for two-phase flow based upon mixed finite elements and nonuniform coarse grids. *Multiscale Model. Simul.*, 5(2):337–363. doi: 10.1137/050634566.

Aarnes, J. E., Krogstad, S., and Lie, K.-A. 2008. Multiscale mixed/mimetic methods on corner-point grids. *Comput. Geosci.*, 12(3):297–315. doi: 10.1007/s10596-007-9072-8.

Alberty, J., Carstensen, C., and Funken, S. A. 1999. Remarks around 50 lines of Matlab: short finite element implementation. *Numer. Algorithms*, 20(2-3):117–137. doi: 10.1023/A:1019155918070.

Alpak, F. O., Pal, M., and Lie, K.-A. 2012. A multiscale method for modeling flow in stratigraphically complex reservoirs. *SPE J.*, 17(4):1056–1070. doi: 10.2118/140403-PA.

Andersen, O., Nilsen, H. M., and Lie, K.-A. 2014. Reexamining CO_2 storage capacity and utilization of the Utsira Formation. In *ECMOR XIV – 14^{th} European Conference on the Mathematics of Oil Recovery, Catania, Sicily, Italy, 8-11 September 2014*. EAGE. doi: 10.3997/2214-4609.20141809.

Cao, H. 2002. *Development of techniques for general purpose simulators*. PhD thesis, Stanford University.

DeBaun, D., Byer, T., Childs, P., Chen, J., Saaf, F., Wells, M., Liu, J., Cao, H., Pianelo, L., Tilakraj, V., Crumpton, P., Walsh, D., Yardumian, H., Zorzynski, R., Lim, K.-T., Schrader, M., Zapata, V., Nolen, J., and Tchelepi, H. A. 2005. An extensible architecture for next generation scalable parallel reservoir simulation. In *SPE Reservoir Simulation Symposium, 31 January–2 Feburary, The Woodlands, Texas, USA*. doi: 10.2118/93274-MS.

Gries, S., Stüben, K., Brown, G. L., Chen, D., and Collins, D. A. 2014. Preconditioning for efficiently applying algebraic multigrid in fully implicit reservoir simulations. *SPE J.*, 19(04):726–736. doi: 10.2118/163608-PA.

Hasan, A., Foss, B., Krogstad, S., Gunnerud, V., and Teixeira, A. F. 2013. Decision analysis for long-term and short-term production optimization applied to the voador field. In *SPE Reservoir Characterization and Simulation Conference and Exhibition, 16-18 September, Abu Dhabi, UAE*. SPE 166027-MS, doi: 10.2118/166027-MS.

Hauge, V. L. and Aarnes, J. E. 2009. Modeling of two-phase flow in fractured porous media on unstructured non-uniformly coarsened grids. *Transport in Porous Media*, 77(3):373–398. doi: 10.1007/s11242-008-9284-y.

Hauge, V. L., Lie, K.-A., and Natvig, J. R. 2012. Flow-based coarsening for multiscale simulation of transport in porous media. *Comput. Geosci.*, 16(2):391–408. doi: 10.1007/s10596-011-9230-x.

Hilden, S. T., Lie, K.-A., and Raynaud, X. 2014. Steady state upscaling of polymer flooding. In *ECMOR XIV – 14^{th} European Conference on the Mathematics of Oil Recovery, Catania, Sicily, Italy, 8-11 September 2014*. EAGE. doi: 10.3997/2214-4609.20141802.

Jørgensen, K. 2013. Implementation of a surfactant model in MRST with basis in Schlumberger's Eclipse. Master's thesis, Norwegian University of Science and Technology.

Killough, J. E. 1995. Ninth SPE comparative solution project: A reexamination of black-oil simulation. In *SPE Reservoir Simulation Symposium, 12-15 February 1995, San Antonio, Texas*. SPE 29110-MS, doi: 10.2118/29110-MS.

Krogstad, S. 2011. A sparse basis POD for model reduction of multiphase compressible flow. In *2011 SPE Reservoir Simulation Symposium, The Woodlands, Texas, USA, 21-23 February 2011*. doi: 10.2118/141973-MS.

Krogstad, S., Hauge, V. L., and Gulbransen, A. F. 2011. Adjoint multiscale mixed finite elements. *SPE J.*, 16(1):162–171. doi: 10.2118/119112-PA.

Krogstad, S., Lie, K.-A., and Skaflestad, B. 2012. Mixed multiscale methods for compressible flow. In *Proceedings of ECMOR XIII–13th European Conference on the Mathematics of Oil Recovery*, Biarritz, France. EAGE. doi: 10.3997/2214-4609.20143240.

Leeuwenburgh, O., Peters, E., and Wilschut, F. 2011. 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*. doi: 10.2118/143576-MS.

Li, X. and Zhang, D. 2014. A backward automatic differentiation framework for reservoir simulation. *Comput. Geosci.*, pages 1–14. doi: 10.1007/s10596-014-9441-z.

Lie, K.-A. 2014. *An Introduction to Reservoir Simulation Using MATLAB*. `http://www.sintef.no/Projectweb/MRST/Publications/`.

Lie, K.-A., Krogstad, S., Ligaarden, I. S., Natvig, J. R., Nilsen, H., and Skaflestad, B. 2012. Open-source MATLAB implementation of consistent discretisations on complex grids. *Comput. Geosci.*, 16:297–322. doi: 10.1007/s10596-011-9244-4.

Lie, K.-A., Natvig, J. R., Krogstad, S., Yang, Y., and Wu, X.-H. 2014a. Grid adaptation for the Dirichlet–Neumann representation method and the multiscale mixed finite-element method. *Comput. Geosci.*, 18(3):357–372. doi: 10.1007/s10596-013-9397-4.

Lie, K.-A., Nilsen, H. M., Andersen, O., and Møyner, O. 2014b. A simulation workflow for large-scale CO_2 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. doi: 10.3997/2214-4609.20141877.

Møyner, O., Krogstad, S., and Lie, K.-A. 2014. The application of flow diagnostics for reservoir management. *SPE J.* accepted, doi: 10.2118/171557-PA.

Møyner, O. and Lie, K.-A. 2014a. The multiscale finite-volume method on stratigraphic grids. *SPE J.*, 19(5):816–831. doi: 10.2118/163649-PA.

Møyner, O. and Lie, K.-A. 2014b. A multiscale two-point flux-approximation method. *J. Comput. Phys.*, 275:273–293. doi: 10.1016/j.jcp.2014.07.003.

MRST 2014. The MATLAB Reservoir Simulation Toolbox, version 2014a. `http://www.sintef.no/MRST/`.

Neidinger, R. 2010. Introduction to automatic differentiation and MATLAB object-oriented programming. *SIAM Review*, 52(3):545–563. doi: 10.1137/080743627.

Nilsen, H. M., Lie, K.-A., and Andersen, O. 2014a. Fully implicit simulation of vertical-equilibrium models with hysteresis and capillary fringe. *submitted*.

Nilsen, H. M., Lie, K.-A., and Andersen, O. 2014b. Robust simulation of sharp-interface models for fast estimation of CO_2 trapping capacity. *submitted*.

Nilsen, H. M., Lie, K.-A., and Natvig, J. R. 2012. Accurate modelling of faults by multipoint, mimetic, and mixed methods. *SPE J.*, 17(2):568–579. doi: 10.2118/149690-PA.

Norne 2012. IO Center – Norne Benchmark Case. `http://www.ipt.ntnu.no/~norne`.

Odeh, A. S. 1981. Comparison of solutions to a three-dimensional black-oil reservoir simulation problem. *J. Petrol. Techn.*, 33(1):13–25. doi: 10.2118/9723-PA.

Raynaud, X., Krogstad, S., and Nilsen, H. M. 2014. Reservoir management optimization using calibrated transmissibility upscaling. In *ECMOR XIV – 14th European Conference on the Mathematics of Oil Recovery, Catania, Sicily, Italy, 8-11 September 2014*. EAGE. doi: 10.3997/2214-4609.20141864.

Sandve, T., Berre, I., and Nordbotten, J. 2012. An efficient multi-point flux approximation method for discrete fracturematrix simulations. *J. Comput. Phys.*, 231(9):3784 – 3800. doi: 10.1016/j.jcp.2012.01.023.

Todd, M. R. and Longstaff, W. J. 1972. The development, testing, and application of a numerical simulator for predicting miscible flood performance. *J. Petrol. Tech.*, 24(7):874–882.

Voskov, D. V. and Tchelepi, H. A. 2012. Comparison of nonlinear formulations for two-phase multi-component eos based simulation. *J. Petrol. Sci. Engrg.*, 82-83(0):101–111. doi: 10.1016/j.petrol.2011.10.012.

Voskov, D. V., Tchelepi, H. A., and Younis, R. 2009. General nonlinear solution strategies for multiphase multicomponent eos based simulation. In *SPE Reservoir Simulation Symposium, 2–4 February, The Woodlands, Texas*. doi: 10.2118/118996-MS.

Younis, R. 2009. *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.

Younis, R. and Aziz, K. 2007. Parallel automatically differentiable data-types for next-generation simulator development. In *SPE Reservoir Simulation Symposium, 26–28 February, Houston, Texas, USA*. SPE 106593-MS, doi: 10.2118/106493-MS.

Zhou, Y., Tchelepi, H. A., and Mallison, B. T. 2011. 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*. SPE 141592-MS, doi: 10.2118/141592-MS.