12

The AD-OO Framework for Reservoir Simulation

In Chapter 7 we showed that combining a fully implicit formulation with automatic differentiation makes it 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 11.2, 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.

When you research new computational methods for reservoir simulation, it is important to have a flexible research tool that enables you to quickly test new ideas and verify their performance on a large variety problems, from idealized and conceptual cases to full simulation setups. The AD-OO framework was introduced to structure our fully implicit simulators and enable us to rapidly implement new proof-of-concept codes also for more complex flow physics. Altogether, the AD-OO framework offers many of the features found in commercial simulators. 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. If you are not interested in code details, you can read Section 12.1 and then jump directly to Section 12.4, which presents a few simulation examples so as to outline the functionality and technical details you need to set up your own black-oil simulation both with and without the use of ECLIPSE input. The sections in between give a somewhat detailed discussion of the most central classes and member functions to demonstrate the general philosophy behind the framework, but do not discuss in detail how boundary conditions, source terms, and wells are implemented. The purpose of these sections is to provide you with sufficient detail to start developing new simulators with new numerics or other types of flow physics.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

12.1 Overview of the Simulator Framework

Looking back at the procedural implementations presented so far in the book, we have introduced a number of data objects to keep track of all the different entities that make up a simulation model:

- a *state object* holding the unknown pressures, saturations, concentrations, inter-cell fluxes, and unknowns associated with 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 net-to-gross as well;
- a sparse *transmissibility vector* (or inner-product matrix), possibly including 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 an optional structure that contains *discrete operators* for differentiation and averaging.

In a procedural code, these data objects are passed as arguments to solvers or functions performing subtasks 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, a fluid object, and optional parameters that describe the drive mechanisms. The transport solvers, on the other hand, require state, grid, fluid, rock properties, and drive mechanisms. Solvers for the fully implicit 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:

```
[p, sW, sG, rs, rv] =  ...
    model.getProps(state, 'pressure', 'water', 'gas', 'rs', 'rv');
```

As we saw in the previous chapter, the criteria by which we choose s_g, r_s, and r_v as primary reservoir variable will vary from cell to cell 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 residual flow equations (and their Jacobian) by a call like

```
eqs = model.getEquations(oldState, newState, dT);
```

With a generic interface like this, you can develop nonlinear solvers that require no specific or very limited knowledge of the physical model. A next step would be to separate the

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

time-stepping strategy and the linear solver from the nonlinear solvers, so that we easily can switch between different methods or replace a generic method by a tailor-made method. Moreover, a critical evaluation of the many simulation examples developed with the procedural approach shows that large fractions of the scripts are devoted to input and output of data, plotting results, etc. Likewise, implementing variations of the same model either results 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 is 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 formulated using generic discretization, averaging, and interpolation operators as discussed previously, and
- *linear solvers* for the solution of the linear system

into different *numerical contexts*. 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 that utilizes some kind of time-control mechanism to run

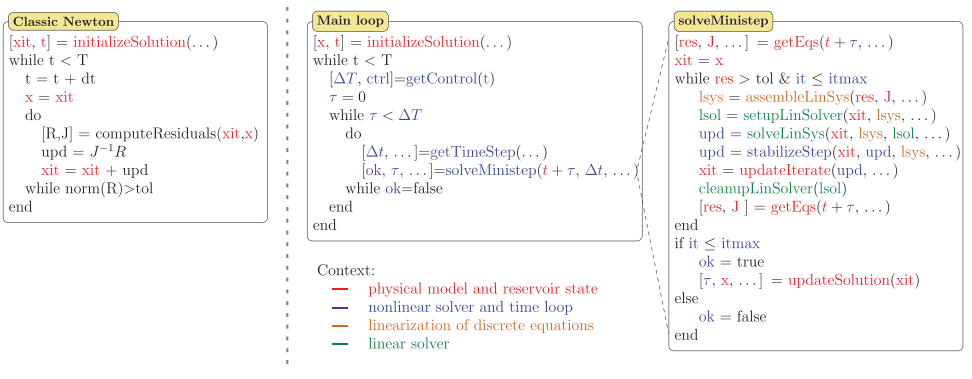

Figure 12.1 Time loop of a simulator for a simple Newton solver (left) and for a more sophisticated solver with stabilization and time-step control (right). For the latter, the iteration has been divided into various numerical contexts.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

the targeted *control steps* of a simulation schedule. (Usually, the simulator should report the solution after each such step.)

In this setup, the *physical model* is responsible 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 Jacobian 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 iteration 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 MRST's core functionality 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 it is able to perform the generic operations we just outlined; we will come back to model classes in the next section. The time-step selectors can implement simple heuristic algorithms that limit how much solutions can change during an iterative step, e.g., as in the Appleyard and modified Appleyard chop [271] used in commercial simulators. The linear solver class provides an interface for

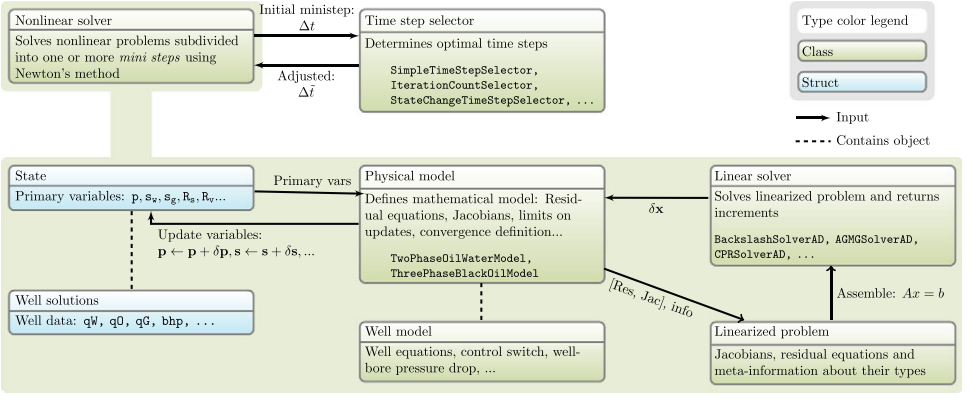

Figure 12.2 Data diagram for a single nonlinear solve with time-step estimation in the AD-OO framework.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

various solvers including MATLAB's default backslash solver as well as a state-of-the-art, constrained pressure residual (CPR) preconditioner [304, 305, 119], which can be combined with efficient algebraic multigrid solvers such as the aggregation-based method found in [238], or the more recent AMGCL library [81]. 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. The linear solver class employs this information to help setting up preconditioning strategies that exploit special structures in the problem.

Example 12.1.1 (1D Buckley–Leverett) *To illustrate the use of the nonlinear solver, let us revisit the classical Buckley–Leverett problem from Example 9.3.4 on page 284 and Section 10.3.1. For completeness, we show all code lines necessary to set up this problem in the AD-OO framework (which you also can find in* `adBuckleyLeverett1D.m` *in the* `ad-core` *module). First, we construct the reservoir and water–oil fluid data with constant shrinkage factors:*

```
mrstModule add ad-core ad-props ad-blackoil
G     = computeGeometry(cartGrid([50, 1, 1], [1000, 10, 10]*meter));
rock  = makeRock(G, 1*darcy, .3);
fluid = initSimpleADIFluid('phases', 'WO', 'n', [2 2]);
```

We then create a two-phase black-oil model (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 water–oil model 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();
```

By default, the solvers employs MATLAB's standard backslash operator to solve the linear problems. The time-step selector employs a simplified version of the time-chop strategy outlined in Section 10.2.2 to run the specified control steps: time steps are chopped in two if the Newton solver requires more than 25 iterations, but no attempt is made to increase successful substeps during a control step. The setup is the same as for the SPE 1 test case 11.8.1 on page 399, except that we then used inexact line search to stabilize the

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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.

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;
solver.verbose = true;
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 an optional input argument that sets up dynamic plotting after each step in the simulation, e.g., to monitor 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 that we 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 requires 7 iterations, whereas the next 15 steps only use 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 runtime so far and uses the iteration history to estimate the total runtime. A total runtime of 7 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.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

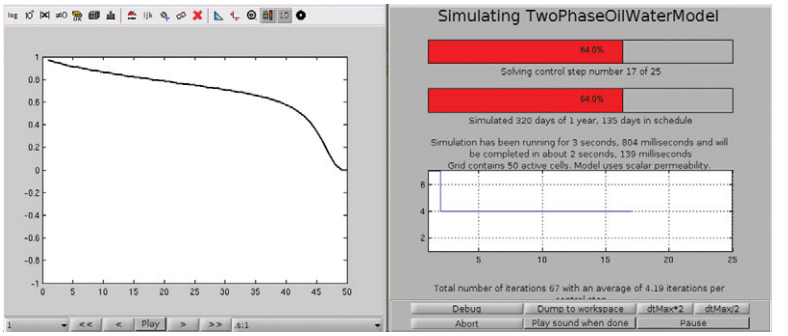

Figure 12.3 The default graphical user interface in the generic simulation loop reports simulation progress and enables simple computational steering.

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 result 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 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 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 set the option* `'storeInMemory'`*.*

You can now use the result handler as a standard cell array, and you need not really know where the results are stored. You can for instance plot the results through the same GUI we used in the previous chapter

```
plotToolbar(G, handler, 'field', 's:1','lockCaxis', true, 'plot1d', true);
```

If you at a later time wish to retrieve the data, you simply 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:

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
handler = ResultHandler('dataFolder', 'ad-bl1d');
m = handler.numelData();
states = cell(m, 1);
for i = 1:2:m
   states{i} = handler{i};
end
```

You can delete the stored data 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 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. Instead, they provide generic quantities and access mechanisms, which we will discuss in more detail shortly. We can then develop specific models by expanding the generic models with specific flow equations and fluid and rock properties. The `ad-core` module also contains classes implementing well models: either simple Peaceman wells of the type we discussed in Sections 4.3.2, 5.1.5, and 11.7.1, or advanced multisegment well models from Section 11.7.2 that give more accurate representation of deviated, horizontal, and multilateral wells with advanced completions and inflow control devices.

The `ad-blackoil` module implements black-oil models, as discussed in Chapter 11, utilizing functionality from the `deckformat` module to read ECLIPSE input decks and from `ad-props` to 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.

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 [215, 216]. The `blackoil-sequential` module implements sequential (fully implicit) solvers for the same set of equations as in `ad-blackoil`, based on a fractional

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

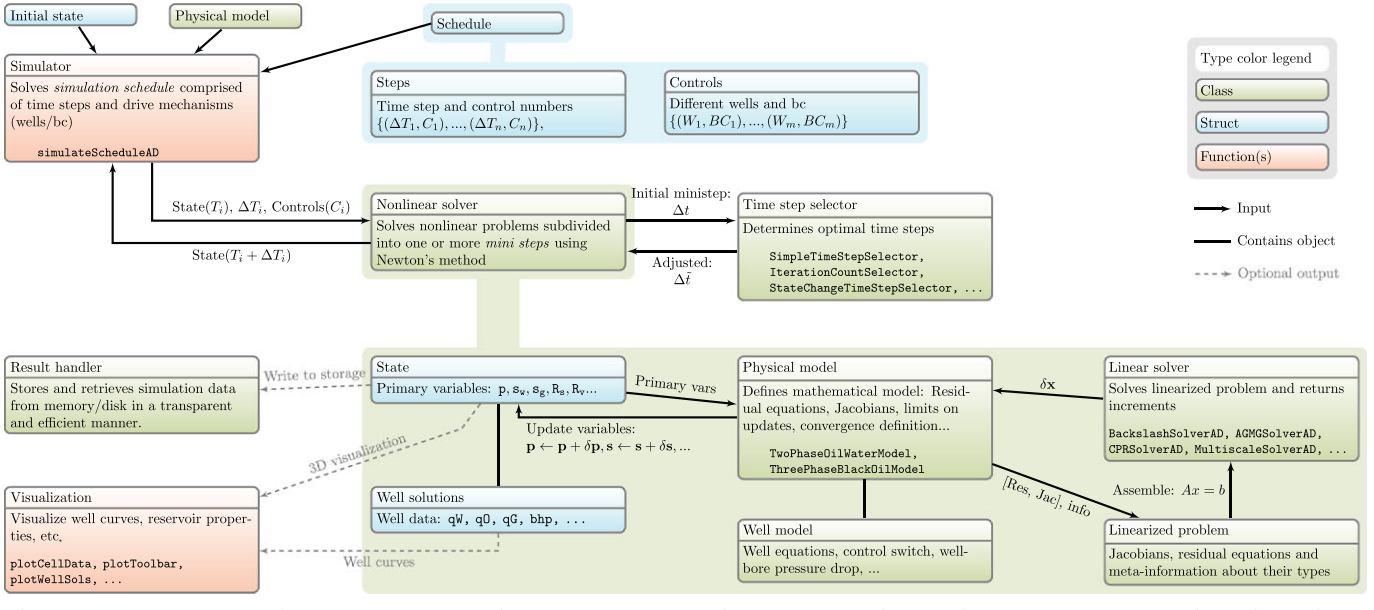

Figure 12.4 *Overview of how components in the AD-OO framework are organized to implement a generic simulator loop (here shown for black-oil equations).*

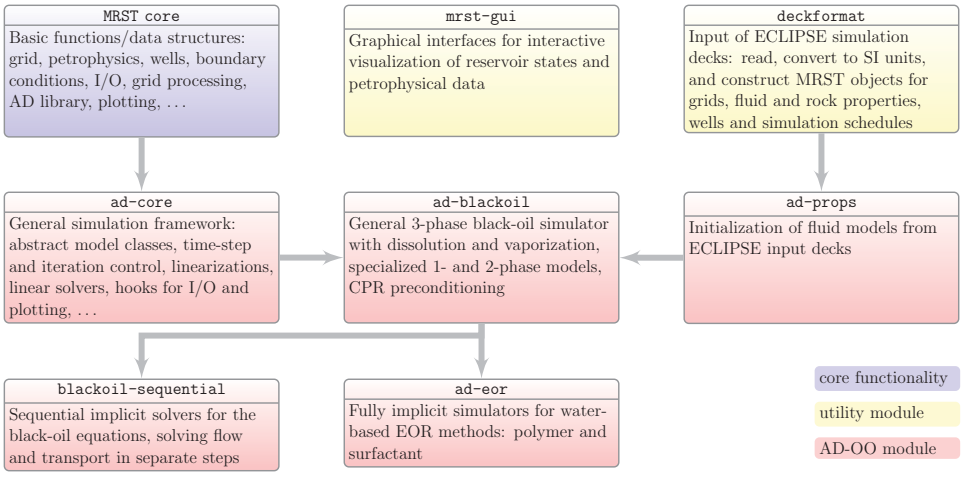

Figure 12.5 Modules used to implement black-oil models in the AD-OO framework.

flow formulation wherein pressure and transport are solved as separate steps and thus are represented as separate models.

The `ad-eor` module [30] implements extensions of the black-oil equations to water-based EOR methods like polymer and surfactant flooding and the `co2lab` module [19] uses inheritance from the `black-oil` module to implement specialized simulators for CO_2 sequestration. The black-oil family also includes modules for solvents and routines for solving optimal control problems based on the AD-OO framework. In addition, MRST contains other AD-OO modules for *compositional* simulators [212], geomechanics, and coupled geomechanics–flow simulations that do inherit from `ad-blackoil`, and more modules are in the making. These are all (unfortunately) outside the scope of this book.

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 *non-linear 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 signifying if the model equations are linear (and hence can be solved by a

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

single matrix inversion), and a flag determining how verbose the member functions should 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 lowercase letter.

Model Equations and Physical Properties

The basic class has several member functions that implement 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 discuss this in more detail in Section 12.3. Input argument `forces` represents driving forces that are currently active. 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 are set by

```
forces = model.getDrivingForces(control);
```

The function creates a cell array that lists 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()
```

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

In addition, the class has utility functions for interacting with the `state` structure in a consistent manner. This includes querying values for 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 function arguments will only work for instances of `ReservoirModel` or derived classes that specify the variables `pressure` and `s`. You can check whether a variable is defined or not as follows

```
[fn, index] = model.getVariableField(name)
```

which produces the field name and column index that will extract the correct variable from the state object. The function produces 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, it 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 on page 399, 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 that provides information such as 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 enabling 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 with a given increment. The following code updates the pressure value from 10 to 110:

```
state = struct('pressure', 10);
state = model.updateStateFromIncrement(state, 100, problem, 'pressure')
```

We can also restrict the maximum relative changes, e.g., to be at most 10%

```
state = model.updateStateFromIncrement(state, 100, problem, 'pressure', .1)
```

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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 on 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 is returned. Otherwise, an error should occur. This function does not have meaning for a `PhysicalModel`, but should be implemented in derived classes. Likewise, there is a function to validate the state for use with the model

```
model.validateState(state)
```

The function should check that required fields are present and of the right dimensions, and if missing fields can be assigned default values, return `state` 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]);
```

here called for a model of the derived `ReservoirModel` class to check 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. (How this is done is 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 adjoint equations boils down to running a backward simulation with a set of linearized equations. `PhysicalModel` also offers a generic update function `solveAdjoint` that computes a single step for the corresponding adjoint equations.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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

   % Iteration parameters
   dpMaxRel, dpMaxAbs, dsMaxAbs, maximumPressure, minimumPressure,
   useCNVConvergence, toleranceCNV, toleranceMB
   :
```

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);
  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.operators = setupOperatorsTPFA(G, model.rock, ..)
end % call for construction: ReservoirModel(G, rock, fluid, ...)
```

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 an absolute change of 0.2.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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(strcmpi(model.getSaturationVarNames, name));
       fn = 's' ;
    :
```

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. Let us consider saturation as an example. Saturation is stored in the field `state.s`, but this vector may have one, two, or three entries per cell, depending upon the number of phases present. To access individual phase saturations, the class implements 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 is 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 the code continues to work if we later decide to change the name of the saturation variable, or 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.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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 `relPermWO`, `relPermOG`, `relPermWG`, and `relPermWOG` functions discussed in Section 11.3, which in turn are interfaces to the more basic `krW`, `krO`, `krG`, `krOG`, and `krOW` functions of fluid objects in MRST. The generic reservoir model neither implements nor offers any interface for PVT behavior, since this is 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 did not implement any. `ReservoirModel` 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 because each well is 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 in principle can contain models of all 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

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
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
```

The generic operation of adding 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 done in the `insertWellEquations` function. The same holds for source terms and boundary conditions. If you are more interested, you should consult the code for further details.

Updating States and Models

The computation of linearized increments in a reservoir model is inherited from the `PhysicalModel` class without any special adaptions or extensions, since it follows the exact same algorithm as in a generic Newton loop. On the other hand, the way these increments are used to update reservoir states differs 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 find all necessary details in the code.

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 n_{ph}'th increment is set as the negative sum of all the other increments; i.e., values added to the first $n_{ph} - 1$ variables must be subtracted from the saturation of the last 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 $S_\alpha \leftarrow S_\alpha / \sum_\alpha S_\alpha$.

We update pressure 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 Models and States

You may recall that `PhysicalModel` defined functions for validating models and states but did not implement any specific behavior. The reservoir model checks that the model instance contains a facility model, and if not, instantiates one such object with default values:

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
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 this case, we extract the well description and pass it 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);
```

To be consistent, the state should contain one unknown pressure and the correct number of phase saturations in each cell. Once we have ensured that the reservoir states meets the requirements, 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. Unlike the two previous abstract classes, 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

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

vaporized oil, respectively. The corresponding data fields `rs` and `rv` are defined in the `getVariableField` function. We also extend `validateState` to check that the reservoir state contains a `rs` and/or a `rv` field if dissolution and/or vaporization effects are active in a specific model instance. The class also defines the maximum absolute/relative increments allowed for r_s and r_v. By default, the constructor sets up a dead-oil model containing the aqueous, gaseous, and oleic phases, but with no dissolution and vaporization effects.

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 is configured. We have already discussed the essential code lines contained in this function, but as stated in Chapter 11, 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 to be used by CPR-type preconditioners, which we outline on page 444.

The `ad-blackoil` module also implements three special cases of the general black-oil model: a single-phase water model and two-phase oil–water and oil–gas models. Since most functionality is inherited from the general model, these special cases are implemented quite compactly, for instance:

```
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 = merge_options(model);
  end
```

Residual equations are evaluated 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
```

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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 11.6. 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.

```
sO  = 1 - sW;
[krW, krO] = model.evaluateRelPerm({sW, sO});
[pvMult, transMult, mobMult, pvMult0] = getMultipliers(model.fluid, p, p0);
krW = mobMult.*krW; krO = mobMult.*krO;
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.*bO.*sO - pvMult0.*bO0.*sO0 ) + s.Div(bOvO);
```

In all codes discussed so far, we have included wells 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.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

12.2.4 Models of Wells and Production Facilities

The AD-OO framework has primarily been developed to study hydrocarbon recovery, CO_2 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 *reservoir* models are therefore assumed to have an entity representing this flow communication between the surface and the subsurface. If you only want to describe flow in the porous medium, you can set the well model to be void, but it must still be present.

Facility Models

The well models discussed earlier in the book apply to a single well (see Section 11.7). Wells are sometimes operated in groups that are subject to an overall control. The corresponding well models must hence be coupled. As an example, 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 or stock tank. 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 models are implemented in MRST.

Peaceman-Type Well Models

The inflow performance relation, $q = J(p_R - p_{wb})$, discussed in Sections 4.3.2 and 11.7.1, 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 (bhp). 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 we briefly introduced in Section 11.7.2. In these models, the flow of fluids from the sandface to the well head is

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

described in terms of a set of 1D flow equations on a general flow network. In ECLIPSE [270], the network must assume 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.

A Word of Caution about 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, you can find all necessary details in the actual code, but 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 orders-of-magnitude fewer unknowns than the number of states defined over the reservoir grid. The vectorized AD library in MRST is primarily designed to perform well for long vectors and large sparse matrices and is not necessarily efficient for scalars and short vectors. To keep the computational efficiency when working with individual wells, it has proved 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 go through the three different types of classes the AD-OO framework uses to solve the discrete equations, motivate their design, and briefly describe their key functionality. If you are interested in developing your 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 that represents the linearization of the whole reservoir model. To this end, the class contains the residual equations evaluated for a given specific state along with meta-information about the equations and the primary variables they are differentiated with respect to. We refer to this 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:

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
classdef LinearizedProblem
properties
    equations, types, equationNames
    primaryVariables
    A, b
    state
    dt, iterationNo, drivingForces
end
```

The cell array `equations` contains 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` contain the names of equations and primary variables; these are used, e.g., by the linear solvers and for reporting convergence. The sparse matrix `A` represents the linear system and the vector `b` holds the right-hand side. 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 by the member function

```
[A, b] = problem.getLinearSystem();
```

The function will check whether the linear system already exists, and if not, it will assemble it 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
```

In our previous AD solvers, the equations were always AD objects, and we could assemble the linearized system simply by concatenating the AD variables vertically. In a general model, we cannot guarantee that all residual equations are AD objects, and hence the vertical concatenation is implemented as a separate function we can overload by specialized implementations. If the residual equations are given as real numbers, this function simply reads

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
function h = combineEquations(varargin)
  h = vertcat(varargin{:});
end
```

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, you can eliminate all variables that are not of a specified type

```
[problem, eliminated] = reduceToSingleVariableType(problem, type)
```

Using this function, you can for instance eliminate all equations that are not posed on grid 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, 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 if the nonlinear solver convergences too slowly. 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 11.8.1 on page 399. The following is an almost complete declaration of the class and its data members

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
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 must call a linear solver and hence has a `LinearSolver` class object; the default is the standard backslash solver. Likewise, we need time-step selection, which in the default implementation does not offer any advanced heuristics for computing optimal time steps. Furthermore, there are limits on the number of times the control steps can be subdivided 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 you should in general not modify this parameter unless you really 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. Obviously, you should never rely on non-converged results, but this behavior may be useful for debugging purposes.

Use of Handle Classes

You 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,

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

so that any changes occuring 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*. Each of these 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 and 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 ministeps, 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 pluggable 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 [271].

The first measure is *mass balance*. When summing residual equations over all cells in the grid, all intercell fluxes 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 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

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

definition of the *mass-balance error*. By convention, these errors are scaled to make them problem-independent,

$$\mathrm{MB}_\alpha = \Delta t \bar{B}_\alpha \left(\sum_i \mathcal{R}_{\alpha,i} \; / \sum_i \Phi_i \right), \tag{12.1}$$

where $\bar{B}_\alpha$ is the formation-volume factor of phase α evaluated at average pressure, $\mathcal{R}_{\alpha,i}$ 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. We say the solution process is 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

$$\mathrm{CNV}_\alpha = \Delta t \bar{B}_\alpha \max_i \left| \frac{\mathcal{R}_{\alpha,i}}{\Phi_i} \right| \tag{12.2}$$

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 in the previous subsection, appropriate selection of time step is crucial to ensure low runtimes and good convergence in the nonlinear solver. We have already encountered several selection schemes that use experience from previous time steps 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:

- 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, whereas more than ten iterations indicate problems with time step or model.
- How does a specified control 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:

```
classdef SimpleTimeStepSelector < handle
properties
   history, maxHistoryLength
   isFirstStep, isStartOfCtrlStep
   previousControl, controlsChanged,  resetOnControlsChanged
   firstRampupStep, firstRampupStepRelative
   maxTimestep, minTimestep
   maxRelativeAdjustment, minRelativeAdjustment
   stepLimitedByHardLimits
```

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

The 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 if 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. This 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. (By default, this 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` attempts to ensure that certain properties of the state change at prescribed 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 [58] 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 simple selector 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:

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
classdef LinearSolverAD < handle
 properties
     tolerance, maxIterations
     extraReport, verbose
     replaceNaN, replacementNaN, replaceInf, replacementInf
 end
```

The intention of the class is to provide a general interface to all types of linear solvers, both direct and iterative. 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 for 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 in Section 12.3.1. Next, we call the main member function `solveLinearSystem`, which is nonfunctional in the base class and must be implemented in concrete subclasses. The linear solution is then passed back to the `LinearizedProblem` class for potential postprocessing. Postprocessing is not 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. 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 a special function that reduces a problem to cell

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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 nonzero 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 memory consumption and runtime increase rapidly with model sizes. MATLAB's direct solvers should therefore only be used for models having at most a few tens of thousands of unknowns.

GMRES with ILU Preconditioner

To get a more scalable solver, you need to use an iterative method. Iterative methods used today to solve large-scale, sparse systems are usually of the Krylov subspace type that seek to find an approximation to the solution of the linear system $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ in the Krylov space $\mathcal{K}$ formed by repeatedly applying the matrix $\boldsymbol{A}$ to the residual $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$, where $\boldsymbol{x}_0$ is some initial guess:

$$\mathcal{K}_\nu = \mathcal{K}_\nu(\boldsymbol{A}, \boldsymbol{r}_0) = \text{span}\{\boldsymbol{r}_0, \boldsymbol{A}\boldsymbol{r}_0, \boldsymbol{A}^2\boldsymbol{r}_0, \ldots, \boldsymbol{A}^{\nu-1}\boldsymbol{r}_0\}.$$

One widely used example is the generalized minimal residual (GMRES) method, which seeks the approximate solution as the vector $\boldsymbol{x}_\nu$ that minimizes the norm of the ν'th residual $\|\boldsymbol{r}_\nu\| = \|b - \boldsymbol{A}\boldsymbol{x}_\nu\|$. Krylov vectors are often linearly dependent, and hence GMRES employs a modified Gram–Schmidt orthogonalization, called Arnoldi iteration, to find an orthonormal basis for $\mathcal{K}_\nu$.

The GMRES iteration is generally robust and works well for nonsymmetric systems of linear equations. Iterative solvers are nonetheless only efficient if the condition number of the matrix is not too high. Black-oil models often give linear systems that have quite bad condition numbers. To remedy this, it is common to use a *preconditioner*. That is, instead of solving $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$, we first construct a matrix $\boldsymbol{B}$ that is inexpensive to invert. Expanding, we have $\boldsymbol{b} = \boldsymbol{A}\boldsymbol{B}^{-1}\boldsymbol{B}\boldsymbol{x} = (\boldsymbol{A}\boldsymbol{B}^{-1})\boldsymbol{y}$, which is less expensive to solve if $\boldsymbol{A}\boldsymbol{B}^{-1}$ has better condition number than $\boldsymbol{A}$.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

Herein, we apply an incomplete LU factorization (ILU) method. The true 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 $\boldsymbol{L}$ and $\boldsymbol{U}$ such that $\boldsymbol{LU} \approx \boldsymbol{A}$. When used as a preconditioner, it is common to construct $\boldsymbol{L}$ and $\boldsymbol{U}$ 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 generally consist of a mixture of different variables and can 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, 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);
```

We eliminate well equations (and other non-cell equations) by use of member functions from the `LinearSolverAD` base class, which in turn relies on block-Gaussian algorithms

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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 [305, 119]. To explain the method, we start by writing the equation for the Newton update in block-matrix form

$$-\begin{bmatrix} \boldsymbol{J}_{pp} & \boldsymbol{J}_{ps} \\ \boldsymbol{J}_{sp} & \boldsymbol{J}_{ss} \end{bmatrix} \begin{bmatrix} \Delta \boldsymbol{x}_p \\ \Delta \boldsymbol{x}_s \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}_p \\ \boldsymbol{R}_s \end{bmatrix}.$$

Here, 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 Jacobian matrix blocks read $(\boldsymbol{J}_{m,\ell})_{i,j} = \partial \boldsymbol{R}_{\ell,i} / \partial \boldsymbol{x}_{m,j}$, 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 achieved by modifying the fully implicit system to the form

$$-\begin{bmatrix} \boldsymbol{J}^*_{pp} & \boldsymbol{J}^*_{ps} \\ \boldsymbol{J}_{sp} & \boldsymbol{J}_{ss} \end{bmatrix} \begin{bmatrix} \Delta \boldsymbol{x}_p \\ \Delta \boldsymbol{x}_s \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}^*_p \\ \boldsymbol{R}_s \end{bmatrix},$$

where the off-diagonals of the coupling matrix $\boldsymbol{J}^*_{ps}$ are small, so that $\boldsymbol{J}^*_{pp} \Delta \boldsymbol{x}_p = \boldsymbol{R}^*_p$ resembles an incompressible pressure equation. Because this CPR pressure equation is almost elliptic, we can use on of the many efficient solvers developed for Poisson-type equations to compute an approximate increment $\Delta \boldsymbol{x}_p$ 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 employs an IMPES-like reduction (plus dynamic row sums from [119]), in which we determine appropriate weights w_c so that the accumulation terms of the residual equations cancel

$$\sum\nolimits_c w_c \mathcal{A}_c = 0, \quad \mathcal{A}_{c,i} = \left(\phi \sum\nolimits_\alpha b_\alpha S_\alpha x_{c,\alpha}\right)_i^{n+1} - \left(\phi \sum\nolimits_\alpha b_\alpha S_\alpha x_{c,\alpha}\right)_i^{n}.$$

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 $\boldsymbol{R}^*_p = \sum_c w_c R_c$. 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 $\boldsymbol{J}^*_{pp}$ 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.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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 only two levels: the fine level on which we seek the unknown solution $\boldsymbol{x}$ and a coarser level on which we seek another unknown vector $\tilde{\boldsymbol{x}}$ having much fewer elements. To move between the two levels, we have a prolongation operator $\boldsymbol{P}$ and a restriction operator $\boldsymbol{Q}$. Then, the iterative method can be written as follows:

$$\boldsymbol{x}^* = \boldsymbol{x}_\nu + \boldsymbol{S}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_\nu),$$
$$\boldsymbol{x}_{\nu+1} = \boldsymbol{x}^* + \boldsymbol{P}\boldsymbol{A}^{-1}\boldsymbol{Q}(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}^*).$$

Here, $\boldsymbol{S}$ is a so-called smoother, which typically consists of a few steps of an inexpensive iterative solver (like Jacobi's method or ILU(0)). For a multilevel method, this procedure is performed recursively on a 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 is to select the coarse levels algebraically and build the operator $\boldsymbol{P}$ by analyzing the matrix coefficients of $\boldsymbol{A}$. The restriction operator is then chosen as the transpose of the interpolation matrix, $\boldsymbol{Q} = \boldsymbol{P}^T$, and the coarse matrix is $\boldsymbol{A}_c = \boldsymbol{P}^T\boldsymbol{A}\boldsymbol{P}$.

The `AGMGSolverAD` class is an interface to the aggregation-based algebraic multigrid (AGMG) solver [238, 15], which builds prolongation operators using recursive aggregates of cells with constant interpolation so that the operator on each level is as sparse as possible to avoid that the coarse matrices $\boldsymbol{A}_c$ become more dense the more one coarsens. AGMG was originally released as free open-source, but recent versions are only freely available for academic users. The main advantage of AGMG, beyond a simple MATLAB interface, is that it has low setup cost and low memory requirements. In my experience, it is easy to integrate and 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.

AGMG can also be run in two stages: a setup stage that builds the coarse hierarchies and the prolongation operators, and a solution stage that employs the prolongation operators to solve a specific problem. This way, you can hope to reduce computational costs by reusing the setup stage for multiple subsequent solves. This behavior can be set up 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
```

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
    [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 with MRST if the solver has a standard MATLAB interface.

Interface to External Linear Solvers

Recently, we added a new module called `linearsolver` to MRST, whose purpose is to offer a uniform interface to other external solvers on the form:

```
x = solver(A, b, tolerance, iterations)
```

So far, the module implements support for AMGCL [81], which is a recent and highly efficient header-only C++ library for solving large sparse linear systems with algebraic multigrid methods [296, 285]. AMGCL can either be used as a multigrid solver or as a smoother, and offers a variety of basic building blocks that can be combined in different ways:

- *smoothers*: damped Jacobi, Gauss–Seidel, ILU(0), ILU(k), sparse approximate inverse, and others;
- *coarsening methods*: aggregation, smoothed aggregation, smoothed aggregation with energy minimization, and Ruge–Stüben; and
- *Krylov subspace solvers*: conjugated gradients, stabilized biconjugate gradients, GMRES, LGMRES, FGMRES, and induced dimension reduction.

To use AMGCL as a CPR solver, you can, e.g., set it up as follows:

```
lsolve = AMGCL_CPRSolverAD('block_size', 2, ...
                           'maxIterations', 150, 'tolerance', 1e-3);
lsolve.setCoarsening('aggregation')
lsolve.setRelaxation('ilu0');
lsolve.setSRelaxation('ilu0');
lsolve.doApplyScalingCPR = true;
lsolve.trueIMPES = false;
:
```

At the time of writing, AMGCL is not part of the official MRST release and must be downloaded from a software repository [81], compiled, and integrated via a MEX interface. Initial tests show that AMGCL is significantly faster than AGMG, and installing it is definitely worth the effort if you want to use MRST on models with hundreds of thousand unknowns.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

ECLIPSE Solver: Orthomin and Nested Factorization

The default solver in ECLIPSE is a Krylov subspace method that relies on a set of search vectors defined so that each new vector $\boldsymbol{q}_{\nu+1}$ is orthogonal to all previous vectors $\boldsymbol{q}_0, \boldsymbol{q}_1, \ldots, \boldsymbol{q}_\nu$. Assuming a preconditioner $\boldsymbol{B}$ and an initial guess $\boldsymbol{x}_0$, we start the iteration process by setting $\boldsymbol{q}_0 = \boldsymbol{B}^{-1}\boldsymbol{r}_0$, where $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_0$. The next approximate solution is then defined as

$$\boldsymbol{x}_{\nu+1} = \boldsymbol{x}_\nu + \omega_\nu \boldsymbol{B}^{-1}\boldsymbol{q}_\nu,$$

which inserted into the linear system gives the residual

$$\boldsymbol{r}_{\nu+1} = \boldsymbol{r}_\nu - \omega_\nu \boldsymbol{A}\boldsymbol{B}^{-1}\boldsymbol{q}_\nu = \boldsymbol{r}_\nu - \omega_\nu \hat{\boldsymbol{q}}_\nu.$$

The relaxation parameter ω_ν is optimized by minimizing $\|\boldsymbol{r}_{\nu+1}\|_2$. The resulting *orthomin* method is a special version of the *conjugate gradient* method. For completeness, let us derive the remaining steps. The optimum value for $\omega_{\nu+1}$ is determined by setting

$$\begin{aligned} 0 =& \frac{\partial}{\partial \omega_{\nu+1}} \|\boldsymbol{r}_{\nu+1}\|_2 = \frac{\partial}{\partial \omega_{\nu+1}} \left[\boldsymbol{r}_\nu \cdot \boldsymbol{r}_\nu - 2\omega_\nu \boldsymbol{r}_\nu \cdot \hat{\boldsymbol{q}}_\nu + (\omega_\nu)^2 \hat{\boldsymbol{q}}_\nu \cdot \hat{\boldsymbol{q}}_\nu\right] \\ =& -2\boldsymbol{r}_\nu \cdot \hat{\boldsymbol{q}}_\nu + 2\omega_\nu \hat{\boldsymbol{q}}_\nu \cdot \hat{\boldsymbol{q}}_\nu \qquad \longrightarrow \omega_\nu = \frac{\boldsymbol{r}_\nu \cdot \hat{\boldsymbol{q}}_\nu}{\hat{\boldsymbol{q}}_\nu \cdot \hat{\boldsymbol{q}}_\nu}, \end{aligned}$$

The search vectors are set to lie in the space spanned by $\boldsymbol{r}_{\nu+1}$ and all the previous search vectors, i.e.,

$$\boldsymbol{q}_{\nu+1} = \boldsymbol{r}_{\nu+1} + \sum_{\ell=0}^{\nu} \alpha_\nu^\ell \boldsymbol{q}_\ell,$$

subject to the constraint that $\hat{\boldsymbol{q}}_{\nu+1}$ is orthogonal to all $\hat{\boldsymbol{q}}_k$ for $k \leq \nu$, i.e.,

$$0 = \hat{\boldsymbol{q}}_{\nu+1} \cdot \hat{\boldsymbol{q}}_\ell = \left[\boldsymbol{A}\boldsymbol{B}^{-1}\boldsymbol{r}_{\nu+1} + \sum_{\ell=0}^{\nu} \alpha_\nu^\ell \hat{\boldsymbol{q}}_\ell\right] \cdot \hat{\boldsymbol{q}}_k.$$

Using the fact that $\hat{\boldsymbol{q}}_\ell \cdot \hat{\boldsymbol{q}}_k$ for $\ell, k \leq \nu$ and $\ell \neq k$, we can easily show that

$$\alpha_\nu^\ell = -\frac{(\boldsymbol{A}\boldsymbol{B}^{-1}\boldsymbol{r}_{\nu+1}) \cdot \hat{\boldsymbol{q}}_\ell}{\hat{\boldsymbol{q}}_\ell \cdot \hat{\boldsymbol{q}}_\ell}.$$

This completes the definition of the *orthomin* method.

For grids having a rectangular topology, it is possible to develop a particularly efficient preconditioner called *nested factorization* [24]. As we have seen previously, MRST uses a *global* numbering of the unknowns, in which the numbering of unknowns first runs over cells and then over variables; this to utilize highly efficient vectorization from MATLAB. In a simulator written in a compiled language, it is usually more efficient to use a *local* numbering that first runs over variables and then over cells. With a standard TPFA

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

and single-point upwind-mobility discretization, the resulting system will have a block-heptagonal form,

$$\boldsymbol{A} = \boldsymbol{L}_3 + \boldsymbol{L}_2 + \boldsymbol{L}_1 + \boldsymbol{D} + \boldsymbol{U}_1 + \boldsymbol{U}_2 + \boldsymbol{U}_3$$

where $\boldsymbol{D}$ is a diagonal matrix whose elements are $m \times m$ matrices representing the unknowns per cell. The block matrices $\boldsymbol{L}_i$ and $\boldsymbol{U}_i$ are lower and upper bands representing connections to neighboring cells in logical direction $i = 1, 2, 3$. Experience shows that the numbering should be chosen so that direction 1 corresponds to the direction of highest transmissibilities, which usually is the z-direction; see [271], whose presentation we henceforth follow. The key idea of nested factorization is now to utilize this special structure to recursively define a set of preconditioners, whose action can be computed iteratively by solving block-tridiagonal systems. These can be solved very efficiently by Thomas's algorithm that consists of two substitution sweeps, one forward and one backward. The factorization is built iteratively using the following definition

$$\boldsymbol{B} = (\boldsymbol{P} + \boldsymbol{L}_3)\boldsymbol{P}^{-1}(\boldsymbol{P} + \boldsymbol{U}_3), \tag{12.3}$$

$$\boldsymbol{P} = (\boldsymbol{T} + \boldsymbol{L}_2)\boldsymbol{T}^{-1}(\boldsymbol{T} + \boldsymbol{U}_2), \tag{12.4}$$

$$\boldsymbol{T} = (\boldsymbol{G} + \boldsymbol{L}_1)\boldsymbol{G}^{-1}(\boldsymbol{G} + \boldsymbol{U}_1). \tag{12.5}$$

With a slight abuse of notation, we should understand the terms on the right-hand side as values from the previous iteration. The matrix $\boldsymbol{G}$ is a block-diagonal matrix, which for a fully implicit discretization is defined as,

$$\boldsymbol{G} = \boldsymbol{D} - \boldsymbol{L}_1\boldsymbol{G}^{-1}\boldsymbol{U}_1 - \mathrm{colsum}\big(\boldsymbol{L}_2\boldsymbol{T}^{-1}\boldsymbol{U}_2\big) - \mathrm{colsum}\big(\boldsymbol{L}_3\boldsymbol{P}^{-1}\boldsymbol{U}_3\big). \tag{12.6}$$

This definition ensures that although the solution may not yet be correct, it will have no mass-balance errors. Alternatively, in the IMPES and AIM methods, we can construct $\boldsymbol{G}$ using row sum rather than column sum to ensure accurate pressure.

Let us now look at how the solution proceeds: Starting at the outmost level, we see that to solve $\boldsymbol{B}\boldsymbol{y} = \boldsymbol{r}$, we need to solve $(\boldsymbol{P} + \boldsymbol{L}_3)(\boldsymbol{I} + \boldsymbol{P}^{-1}\boldsymbol{U}_3)\boldsymbol{y} = \boldsymbol{r}$. This can be decomposed as follows

$$\hat{\boldsymbol{y}} = \boldsymbol{P}^{-1}\big(\boldsymbol{r} - \boldsymbol{L}_3\hat{\boldsymbol{y}}\big), \qquad \boldsymbol{y} = \hat{\boldsymbol{y}} - \boldsymbol{P}^{-1}\boldsymbol{U}_3\boldsymbol{y}$$

If $\hat{\boldsymbol{y}}$ is known in the first plane of cells, solving these two equations reduces to a forward elimination and a backward substitution, since the products $\boldsymbol{L}_3\hat{\boldsymbol{y}}$ and $\boldsymbol{U}_3\boldsymbol{y}$ only involve values along planes where the solution is known. In this procedure, we need to invert $\boldsymbol{P}$ and compute vectors $\boldsymbol{z} = \boldsymbol{P}^{-1}\boldsymbol{u}$ along planes of cells. Applying the same idea to (12.4), we obtain

$$\hat{\boldsymbol{z}} = \boldsymbol{T}^{-1}\big(\boldsymbol{u} - \boldsymbol{L}_2\hat{\boldsymbol{z}}\big), \qquad \boldsymbol{z} = \hat{\boldsymbol{z}} - \boldsymbol{T}^{-1}\boldsymbol{U}_2\boldsymbol{z},$$

which we can solve by a forward elimination and a backward substitution provided we know the solution along a first line of cells in each plane. Here, we must compute $\boldsymbol{w} = \boldsymbol{T}^{-1}\boldsymbol{v}$ on each line, which again can be decomposed and solved efficiently as follows:

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

$$\hat{\boldsymbol{w}} = \boldsymbol{T}^{-1}\big(\boldsymbol{v} - \boldsymbol{L}_1\hat{\boldsymbol{w}}\big), \qquad \boldsymbol{w} = \hat{\boldsymbol{w}} - \boldsymbol{T}^{-1}\boldsymbol{U}_1\boldsymbol{w},$$

provided we know $\boldsymbol{G}$ along the lines. From the structure of (12.6), it follows that we can construct $\boldsymbol{G}$ one cell at a time: Starting from a known inverse block $\boldsymbol{G}^{-1}$ in a cell, we can compute the contribution $\boldsymbol{L}_1\boldsymbol{G}^{-1}\boldsymbol{U}_1$ to the next cell in direction 1, and so on. Once $\boldsymbol{G}^{-1}$ is known along a line in direction 1, we can compute the contribution of colsum$\big(\boldsymbol{L}_2\boldsymbol{T}^{-1}\boldsymbol{U}_2\big)$ using the forward-elimination, backward-substitution algorithm for $\boldsymbol{T}^{-1}$. When all the lines in a plane are known, we can compute the contributions from colsum$\big(\boldsymbol{L}_3\boldsymbol{P}^{-1}\boldsymbol{U}_3\big)$ to the next plane in a similar manner. The method may look laborious, but is highly efficient. Unfortunately, it is not applicable to fully unstructured grids.

12.4 Simulation Examples

You have now been introduced to all key components that make up the AD-OO framework. In the following, we 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 example[1] is to demonstrate how to set up a simulator from scratch without the use of input files. To this end, 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 represented by `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, gravity

[1] Complete code: `blackoilTutorialOnePhase.m` in `ad-blackoil`.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

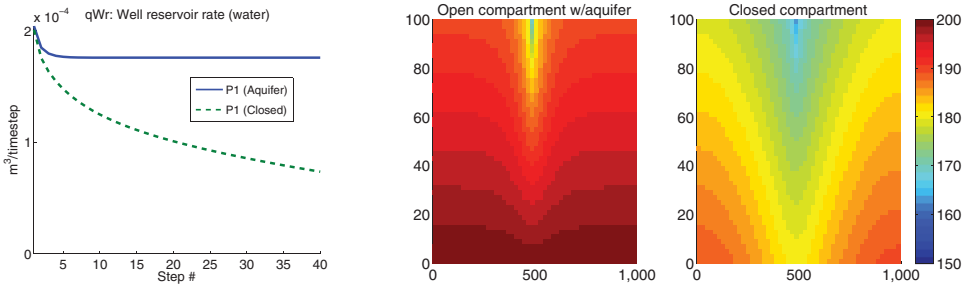

Figure 12.6 Drainage from a single reservoir compartment that is either sealed or supported by an infinite aquifer from below.

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.

```
gravity reset on
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:

```
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);
```

We then create two simulation schedules, 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 is 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 the 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 drawdown at the wellbore will create an elongated pressure pulse that moves fast downward and

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

expands slowly outward. The open compartment quickly reaches 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 is gradually depleted towards the perforation pressure, which in turn causes the production rate to decline towards zero.

12.4.2 An Undersaturated Sector Model

The next example[2] continues in the same vein as the previous: we consider a $1,000 \times 1,000 \times 100$ m^3 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 dissolved gas is assumed to follow that of the SPE 1 benchmark, but unlike in the original setup, water is assumed to be mobile, following a simple quadratic relative permeability.

```
[~, ~, fluid, deck] = setupSPE1(); fluid.krW = @(s) s.^2;
```

Permeability is homogeneous and isotropic, and saturation and pressure are 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 is produced from a single vertical well, placed in the center, operating at a fixed oil rate, but with a lower bottom-hole pressure (bhp) limit:

```
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');
                 W.lims.bhp = 100*barsa;
```

The AD-OO framework assumes no-flow boundary conditions by default, and hence we do not need to specify boundary conditions to simulate closed boundaries. For the other case, we assume that the pressure and fluid composition at the boundary 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 modeling real cases.

[2] Complete code: `blackoilSectorModelExample.m` in `ad-blackoil`

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
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 employs 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.

Turning on the well induces a fast pressure transient that lasts a few days. We do not try to resolve this, but gradually ramp up the time step, as discussed earlier in the book, to improve the initial stability of the simulation:

```
dt = rampupTimesteps(T, 30*day);
```

Specifically, the routine splits the simulation horizon `T` into a number of time step of length 30 days (the last step may be shorter if `T` is not a multiple of the specified time step). It 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 discussed in the previoius subsection

```
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 simulation, i.e., the quantities used to control the production well. After the initial transient, the case with constant pressure boundary settles at a steady state with constant bhp 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 starts 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 4.5 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

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

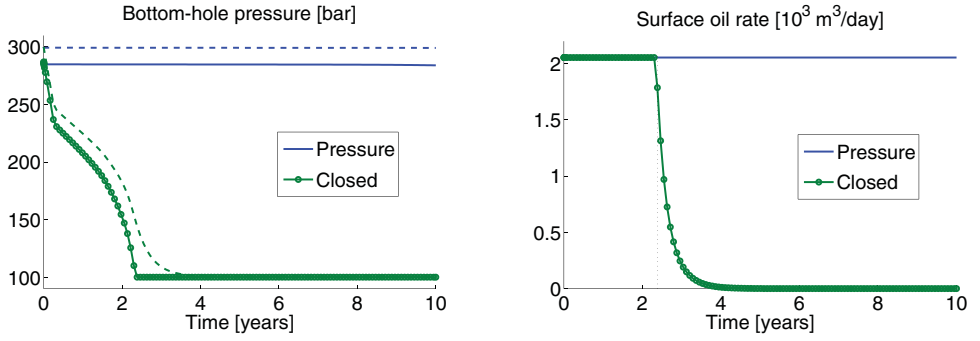

Figure 12.7 Bottom-hole pressure and surface oil rate for the undersaturated sector model; dashed lines show field-average pressure. With closed boundaries, the well control switches from oil surface rate to bottom-hole pressure after 870 days (shown as dotted lines).

reservoir conditions. As the reservoir pressure is gradually lowered, some cells drop below the bubblepoint and free gas is 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 increases and the oil reservoir rate decays 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 causes an accelerating pressure drop as more gas starts flowing 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, the aqueous phase is mobile. As gas is liberated, the mobility of oil is reduced and we also see a significant increase in water production. Topside facilities usually have limited capacity for handling produced water, and the whole production may be reduced or shut down if the water rate exceeds the available capacity. Here, it might 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 reached 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 4.5 years.

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.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

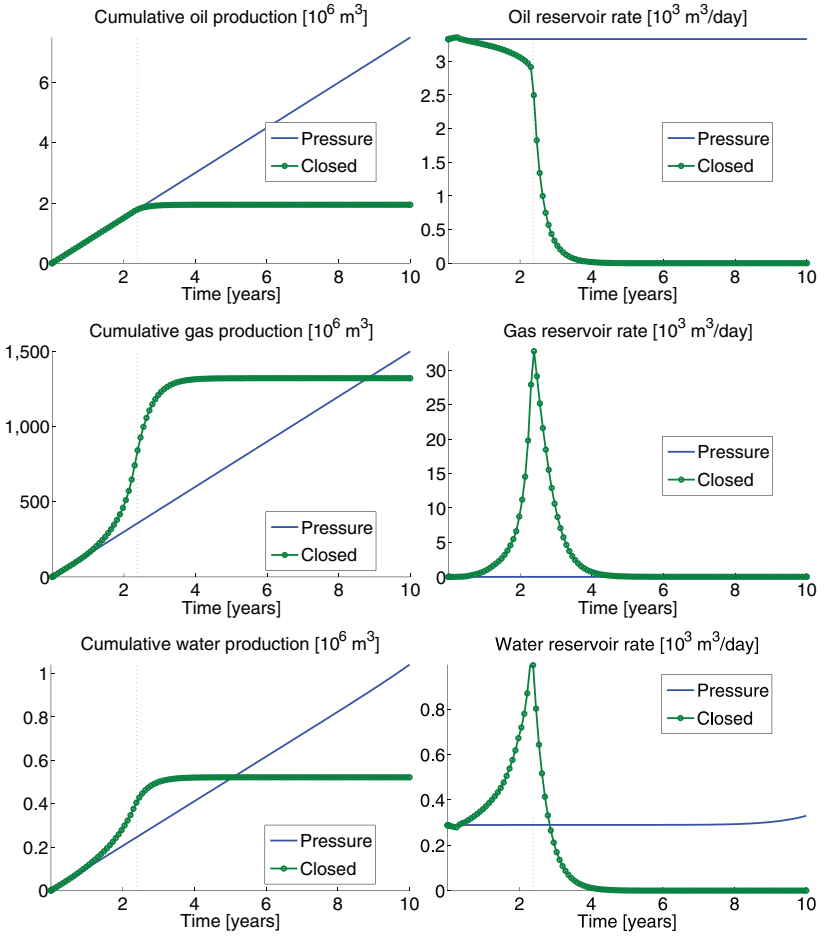

Figure 12.8 Production profiles for the undersaturated sector model. The left column shows cumulative production of the oil, gas, and water components at surface conditions, whereas the right column shows instantaneous phase rates at reservoir conditions.

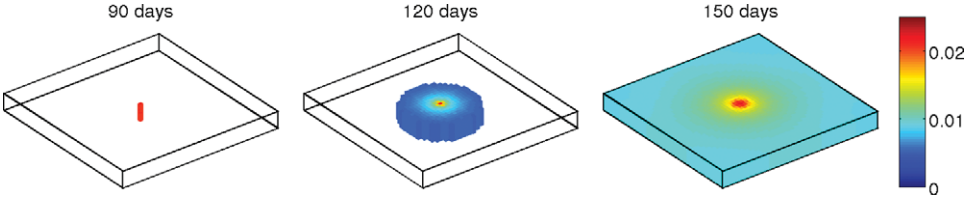

Figure 12.9 Liberation of free gas as the drawdown from the well causes the pressure in the undersaturated oil to drop below the bubblepoint. (The plot shows S_g >5e-3.)

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

COMPUTER EXERCISES

12.4.1 Try to simulate the initial pressure transient, check how long it lasts, and whether it behaves as in the previous example.

12.4.2 The simulation discussed in this section 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.

12.4.3 With constant pressure boundaries, the water rate starts to increase after approximately seven years. Can you explain this behavior?

12.4.4 Gravity was not included in this example. Can you set up multilayer simulations to investigate if gravity has a significant effect on the result?

12.4.3 SPE 1 Instrumented with Inflow Valves

This example[3] demonstrates the use of multisegment wells in MRST. The default well model in MRST assumes that flow in the well takes place on a very short time-scale compared to flow in the reservoir and thus can be accurately modeled as being instantaneous, following the linear inflow performance relationship discussed in Section 11.7.1. Such models are not adequate 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 and enables a more fine-grained representation of the well itself that allows the effects of friction, acceleration, and nonlinear pressure drops across valves and other flow constrictions to be included; see Section 11.7.2.

To illustrate, we revisit the SPE 1 case from Section 11.8.1 and replace the 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.

First, we load the description of the fluid 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`

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

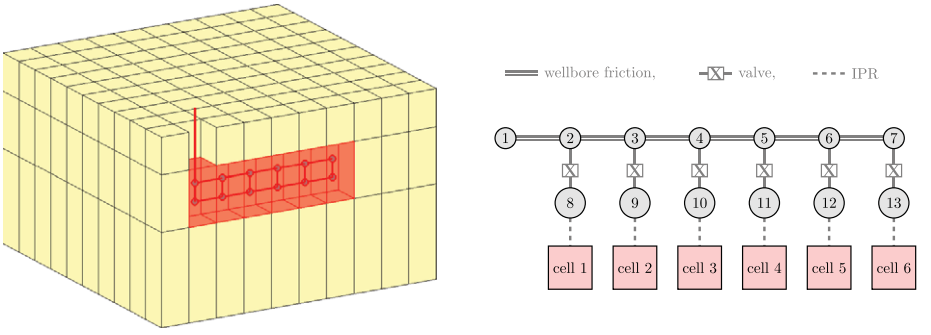

Figure 12.10 Multisegment well for the modified SPE 1 case. Nodes 1–7 represent the wellbore, whereas nodes 8–13 represent the void outside of the valves that separate the wellbore from the sandface.

The model selector examines which phases are present and whether the deck prescribes dissolved gas or vaporized oil, and then determines the specific black-oil subclass that is most suitable for simulating the deck. As we already know, the SPE 1 benchmark describes a three-phase model with dissolved gas but no vaporized oil. The model will thus be an instance of the general `ThreePhaseBlackOilModel` class. Likewise, the schedule converter will parse the specification of all wells and create a suitable schedule object, which here has 120 steps and a single control.

To specify the well models, 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*meter^3/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 = sparse((8:13)', (1:6)', 1, 13, 6);
```

The `cell2node` matrix has dimension equal the number of nodes times the number of reservoir cells perforated by the well. We must also 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);
```

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

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 must set up the flow model for each segment. The first six segments are modeled using the wellbore-friction model (11.29)–(11.31) with roughness $e = 10^{-4}$ m and discharge coefficient $C_v = 0.7$. The next six segments represent valves, whose pressure losses are modeled 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 modeled by the standard inflow performance relationship, i.e., a Peaceman type well model,

```
inj = addWell([], G, rock, 100, 'name', 'inj', 'type', 'rate', 'Comp_i', ...
    [0 0 1], 'val', 2.5e6*meter^3/day,'refDepth', G.cells.centroids(1,3));
```

To compare the effect of instrumenting the production well with inflow valves, we 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 must combine the simple injector and multisegment producer into a facility model since the two wells are represented with different types of well models

```
W = combineMSwithRegularWells(inj, prodMS);
schedule.control.W = W;
model.FacilityModel = model.FacilityModel.setupWells(W);
```

This is normally done automatically by the simulator, but here we do it explicitly on the outside to view the output classes. These classes are practical if we want to incorporate per-well adjustments to tolerances or other parameters. The model classes and the well structures read

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
SimpleWell with properties:

                      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
```

```
MultisegmentWell with properties:

           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
```

```
W is structure with fields:
       cells: 100
        type: 'rate'
         val: 28.9352
           r: 0.1000
           :
   cell2node: []
      connDZ: []
        isMS: 0
       nodes: []
    segments: []
```

```
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]
```

The main difference in the two classes is that the `SimpleWell` model does not have any operators defined. Likewise, this model does not have any representation of cell to node mappings and structures representing the nodes and segments in the `W` structure. However, what MRST uses to distinguish simple and 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 for the instrumented producer

```
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 bhp in the producer and injector, along with 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 must be significantly lower for the multisegment model, compared with the simple well model to sustain the same surface production rate[4]. Looking at the injector bhp, we see that a relatively high pressure is required 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

[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.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

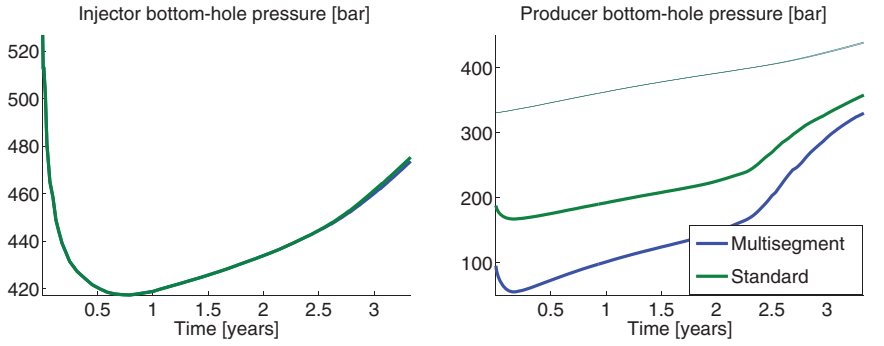

Figure 12.11 Comparison of bottom-hole pressures in the injector (left) and the producer (right) predicted for the modified SPE 1 model with a simple well model and a well model accounting for pressure drop across valves and caused by friction along the horizontal wellbore. Thin lines in the right plot show average-field pressure.

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 bhp starts to increase in the producer after approximately 60 days. On the other hand, 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 20 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 (which still could represent significant income).

Figure 12.13 shows difference in pressure between the wellbore and the perforated cells for both well models. The simple well model assumes instantaneous hydrostatic fluid distribution, and since the producer is completely horizontal, the pressure is identical in all the six points where the wellbore is open to the reservoir. The instrumented well has a significantly larger pressure drawdown near its heel, 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 controls.

Computer exercises

12.4.5 Our setup had no lower limit on the bhp. What happens if bhp in the producer is not allowed to drop below 100 bar or 150 bar?

12.4.6 Rerun the two simulations with producer controlled by a bhp of 250 bar. Can you explain differences in the producer and the bhp of the injector?

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

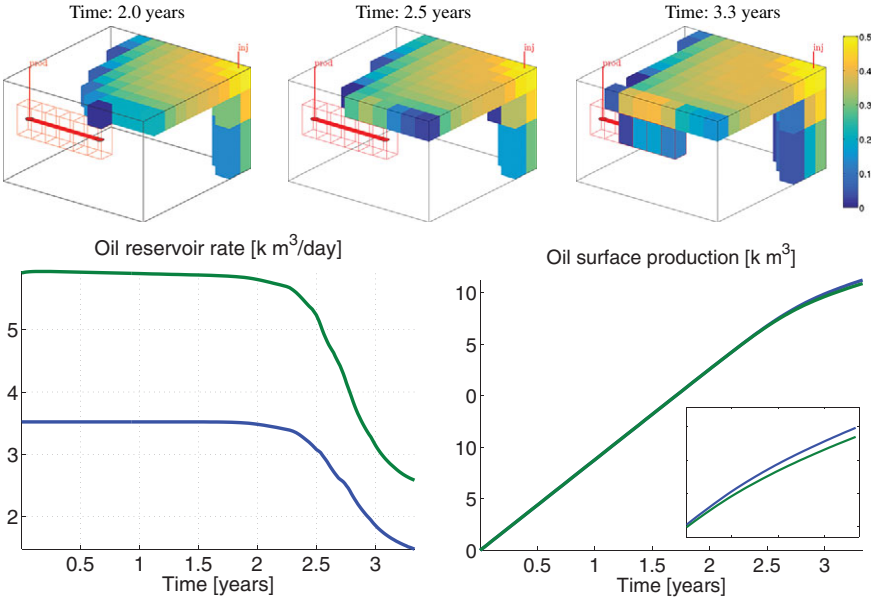

Figure 12.12 Displacement front engulfing the horizontal producer causes decay in oil production. Because of different pressure in the wellbore, the volumetric reservoir rates is largely different in the two models, even though surface rates are almost identical.

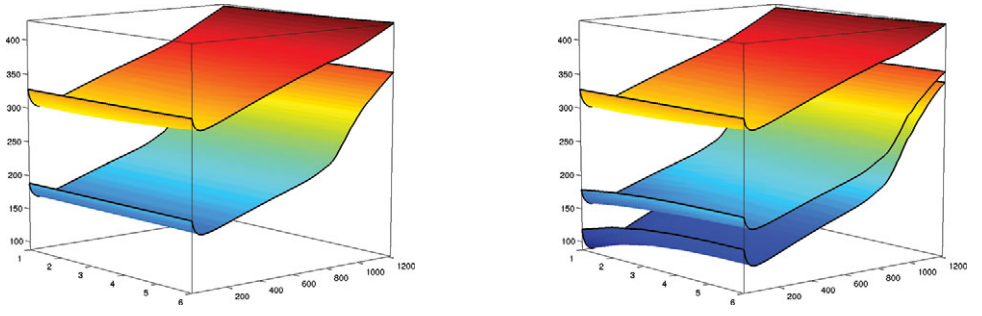

Figure 12.13 Pressure in perforated cells (top) and wellbore (bottom) as function of time for the simple well model (left) and the multisegment well model (right). In the latter plot, the middle surface represents pressure outside of the valves.

12.4.4 The SPE 9 Benchmark Case

In our last example,[5] 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

[5] Complete source code: `blackoilTutorialSPE9.m` in `ad-blackoil`

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

end, we consider the model from the SPE 9 comparative solution project [159], which was posed in the mid-1990s to compare contemporary black-oil simulators. The benchmark consists of a water-injection problem in a highly heterogeneous reservoir described by a $24 \times 25 \times 15$ regular grid with a 10-degree dip 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 11.3.3 and 11.5, 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.

Petrophysical Model and Initial Conditions

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 11.8.1. 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 14 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 variation within each layer. One of the high-permeability layers is represented by two cells in the vertical direction, whereas two of the other layers have the same porosity, altogether giving 12 unique porosity values.

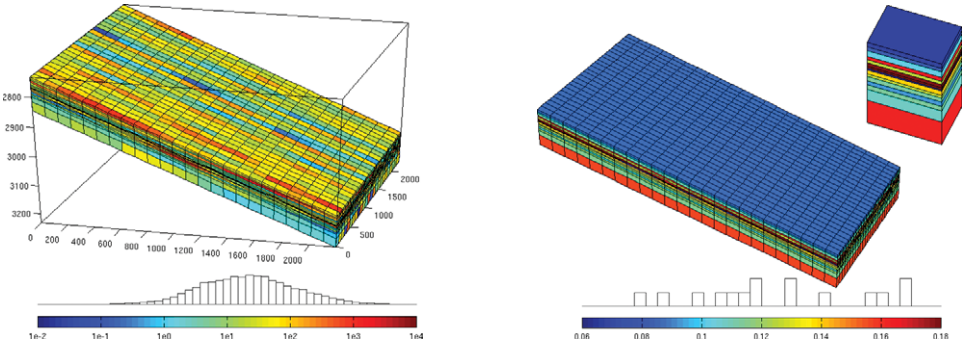

Figure 12.14 Petrophysical data for the SPE 9 model: the left plot shows absolute permeability in units md and the right plot shows porosity.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

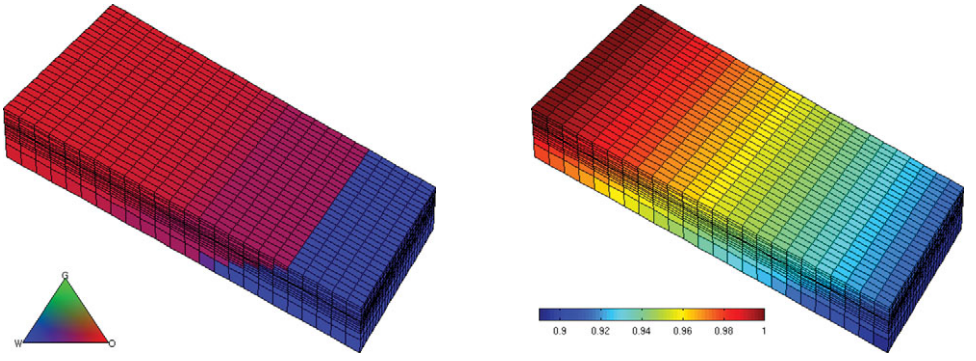

Figure 12.15 Initial fluid distribution inside the reservoir. The left figure shows the saturation, which lies on the two-phase O–W axis in the ternary diagram, whereas the right plot shows the degree of saturation defined as $r_s / R_s(p)$.

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 ($r_s = 0.89 R_s$). We should therefore expect that some free gas will form at the top of the reservoir as pressure drops once the wells start producing.

Wells and Simulation Schedule

The reservoir is produced from 25 vertical wells that initially operate at a maximum rate of 1,500 STBO/D (standard barrels of oil per day). Pressure is supported by a single water injector, operating at a maximum rate of 5,000 STBW/D (standard barrels of water per day) and a maximum bhp of 4,000 psi at reference depth. Between days 300 and 360, the production rate is lowered to 100 STBO/D, and then raised 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.

The simulation schedule consists of three control periods: days 0–300, days 300–360, and days 360–900, 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 bhp and gas/water production vary. In addition to the rate targets, the producers are limited by a lower bhp value of 1,000 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 are given in the `lims` field of each well. For producer number two, this would be `schedule.control(1).W(22).lims`:

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

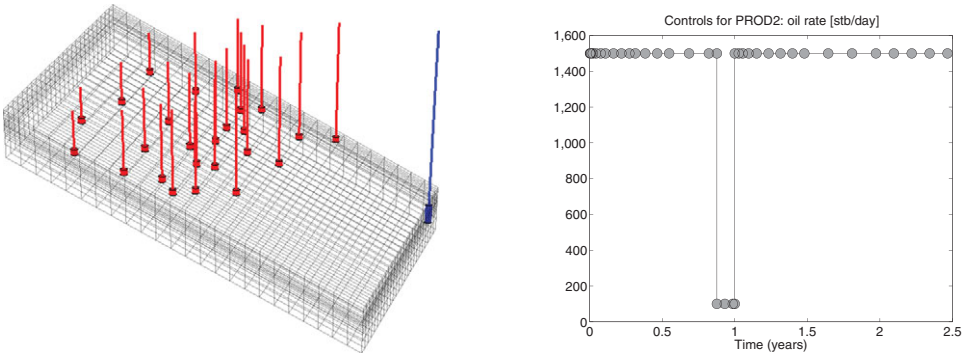

Figure 12.16 Well placement in the reservoir (left) and simulation schedule with rate targets for the producers (right).

```
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 runtimes can be reduced if we instead use a CPR preconditioner. To solve the elliptic pressure subsystem, we 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

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

```
model.dpMaxRel  = .1;
model.dsMaxAbs  = .1;
```

Beyond this, we make no modifications to the standard configuration of the simulator, which employs robust defaults from ECLIPSE.

Running the Simulation

The simulation can now be started by 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 | 2.82e-02 | 1.10e+00 | 5.00e-03 | 1.18e-05 | 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 | 1.24e+00 | 1.47e+06 |
Well PROD26: Control mode changed from orat to bhp.
|    3 |*6.37e-04 | 2.38e-02 | 6.68e-02 |*7.78e-09 | 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-08 | 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 438 for water, oil, and gas. The next three columns report residuals for the inflow performance relationship defined per perforations, whereas the last column reports residual for the control equations defined per well. A star means that the residual is below the tolerance. In the first time step, mass-balance error for oil was the last one to drop below the prescribed value of 10^{-9}. Nine iterations is a quite high number and indicates that there are significant nonlinearity in the residual equations, primarily 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, liberation of gas or other 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 5,000 STBW/day is much

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

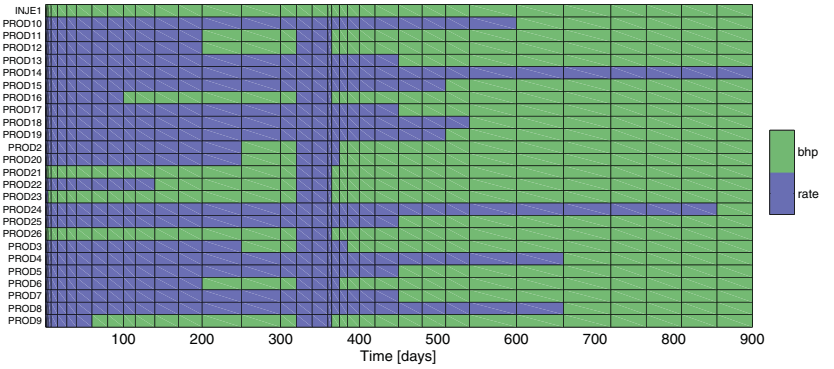

Figure 12.17 Well controls active during the simulation of the SPE 9 benchmark.

higher than what can be sustained with reasonable injection pressures. The injector thus switches 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 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 when rates are reduced by a factor of ten. A smaller drawdown is required to sustain 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 contributes to maintain reservoir pressure and push water into the oil zone to displace oil. Ideally, this would have been a piston-like displacement that gradually pushed the oil–water contact upward, but because of heterogeneity and nonlinear fluid mobilities, water displaces 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 increases monotonically as the reservoir is gradually depleted.

The second drive mechanism is liberation of free gas, which starts near the producers where the pressure drawdown is largest. As pressure continues to decay, more gas is liberated at the top of the formation where the pristine oil phase is closest to a saturated state. The liberated gas forms 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

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

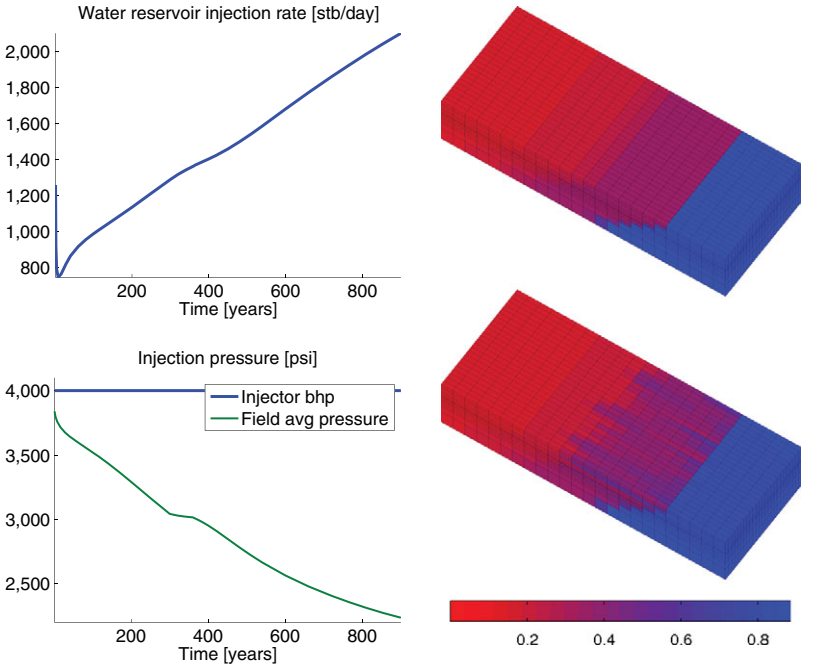

Figure 12.18 Water drive contributing to enhanced production. The left column shows surface injection rate (top) and bhp compared with the average field pressure (bottom). The right column shows the bottom water drive, with initial water saturation at the top and water saturation after 900 days at the bottom.

Figure 12.19. Any produced and free gas must come from the gas initially dissolved in oil, since no gas has been injected. Figure 12.19 reports the amount of dissolved gas initially and after 900 days.

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 that 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. These wells experience increasing surface gas rates until the oil rate is reduced at 300 days. Other wells are of intermediate type and 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 of 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

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

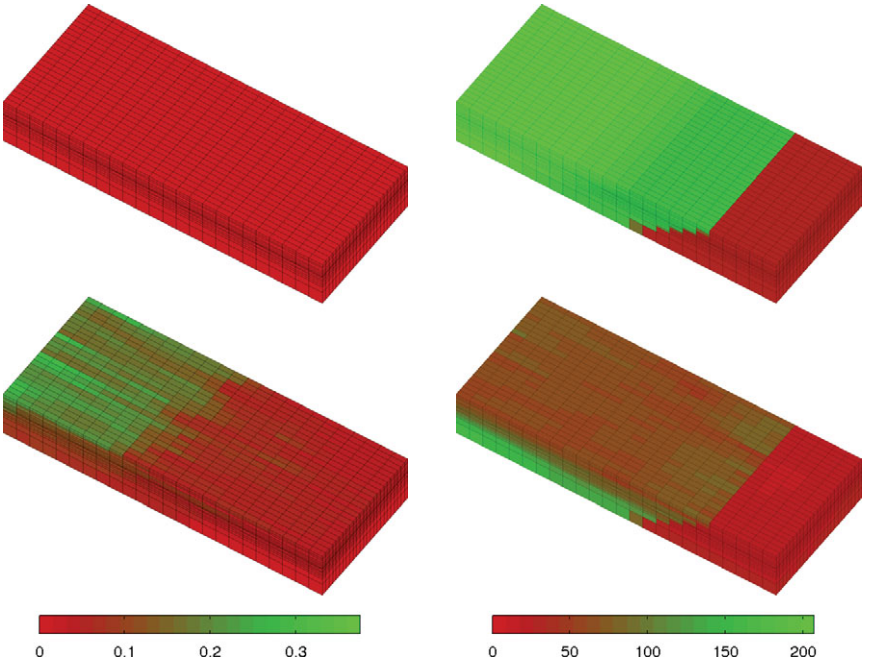

Figure 12.19 Liberation of gas. The left column shows initial gas saturation at the top and gas saturation after 900 days at the bottom. The right column shows the amount of gas dissolved in the oleic phase.

producer. (PROD17 is not shown in the lower-left plot in Figure 12.20.) We note in passing that the plot of bhp 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 matches for the SPE 9 benchmark, given identical input. To this end, we use the same functionality from the `deckreader` module for reading *restart* files as we used in Section 11.8.2:

```
addir = mrstPath('ad-blackoil');
compare = fullfile(addir, 'examples', 'spe9', 'compare');
smry = readEclipseSummaryUnFmt(fullfile(compare, 'SPE9'));
```

We first get the number of report steps and the time at which these are computed

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

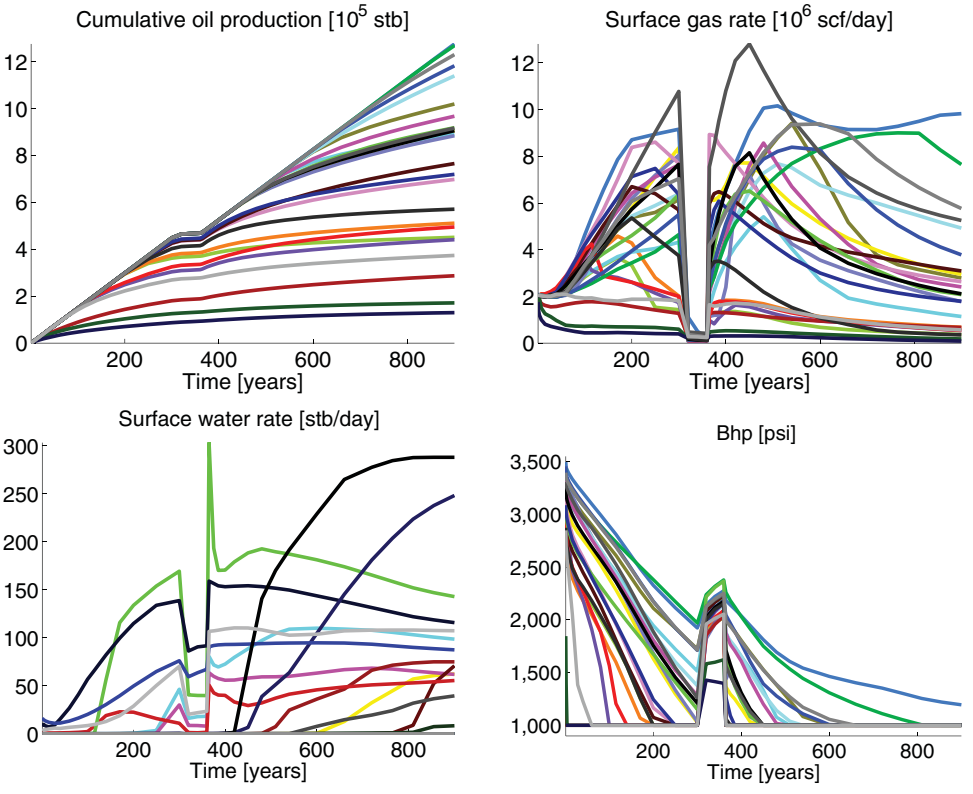

Figure 12.20 Fluid production from the 25 producers.

```
compd = 1:(size(smry.data, 2));
Tcomp =  smry.get(':+:+:+:+', 'YEARS', compd);
```

Let us for instance compare the bhp of PROD13. We extract the corresponding data as follows:

```
comp = convertFrom(smry.get('PROD13', 'WBHP', compd), psia)';
mrst = getWellOutput(wellsols, 'bhp', 'PROD13');
```

and subsequently plot well responses using standard MATLAB commands; see Figure 12.21. The results computed by the two simulators are not identical. If you run the example, you will see that the output from ECLIPSE contains 37 steps whereas MRST reports 35 steps. The reason is that MRST does not include the initial state in its output and that ECLIPSE has a somewhat different time-step algorithm that adds one extra time 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

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

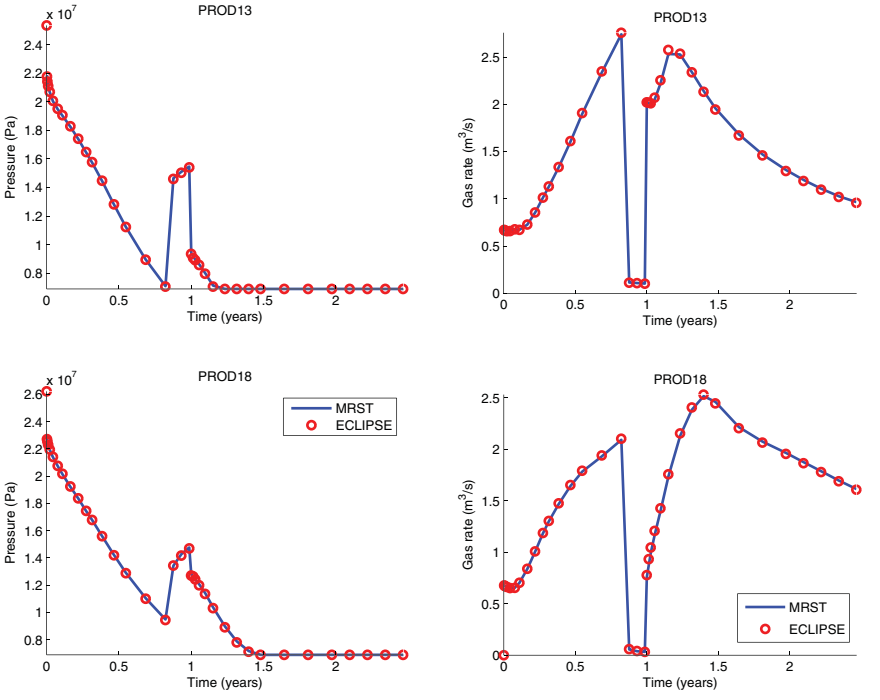

Figure 12.21 Verification of the AD-OO black-oil solver from `ad-blackoil` against ECLIPSE. The left column shows bhp for two different producers, whereas the right column shows surface gas rates.

be almost perfect. Similar matches are also seen for other types of well responses and all the remaining 24 wells.

Bao et al. [30] report similar good match for a set of polymer injection cases including non-Newtonian fluid rheology, In addition, 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 black-oil cases. 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 is also the point where you can contribute by adding new features.

Computer exercises

12.4.7 Set up an animation of water and gas to better understand how the aqueous and gaseous phases displace the oleic phase.

12.4.8 As pointed out on page 466, well PROD17 produces orders of magnitude more water than the other wells. How would shutting this well in from the start influence the overall field production.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

12.4.9 Run the simulation using CPR with AGMG or MATLAB's direct solver and compare with just using the direct solver for the whole system. Are there any noticeable differences in simulation times? Try to install AMGCL and use functionality from the `linearsolver` module discussed on page 446 to see if you can reduce runtime even further.

12.4.10 In some cases, produced gas cannot be sold but must be reinjected. Can you set up such a simulation based on the case discussed in this section? Where would you place the gas injector? (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 the gas is readily available so that whatever amount of gas produced in step n can be injected in step $n + 1$.)

12.5 Improving Convergence and Reducing Runtime

I hope that this and the previous chapters have given you a better understanding of reservoir simulation, and also provided you with sufficient detail about MRST so that you can use the software to perform interesting simulations. Although the software is primarily designed for flexibility and generality, the black-oil simulators implemented by use of the AD-OO framework offer such a comprehensive set of features that they can be used to simulate quite complicated models. Out of the box, MRST may not always be as fast as a simulator written in a compiled language, but with a bit customized setup you can easily make the simulators much more competitive. A key factor to improve computational efficiency of MRST is to replace MATLAB's default direct solver by a modern iterative solver with appropriate multilevel preconditioning as discussed in Section 12.3.4. With such capabilities on board, MRST will typically be a few times slower than ECLIPSE or comparable research simulators like OPM Flow for moderately sized models with $\mathcal{O}(10^5$–$10^6)$ unknowns as long as many-core or multi-node parallelism is not utilized.

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. I end this chapter by briefly discussing how to address such problems. First, I provide a few general recommendations that apply equally well to commercial simulators like ECLIPSE, and then end the section with a few issues that are specific to MRST.

Tweaking Control Parameters for Solvers

Like many other things in the AD-OO framework, default values for the various parameters that control the behavior of the `NonLinearSolver` class largely follow what is suggested for ECLIPSE. These default values have been developed based on many years of experience on a large variety of simulation models and will work well in most cases. You can observe improved performance of your simulation by tuning parameters in specific cases, but it is generally *not recommended* that you change the default values unless you

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

know what you are doing. In particular, it is rarely 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 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 Section 11.8.3.

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 for improving 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 like 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 in 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. (This keyword is not used by MRST.)

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

Issues Specific to MRST/MATLAB

In implementing MRST, we have tried to utilize MATLAB and its inherent vectorization as effectively as possible. However, maintaining both flexibility and efficiency is not always possible, and there are a few sticky points in the implementation of the AD-OO framework that may contribute to increased computational time. For vectors with a few hundred or thousand unknowns, the computational overhead in the AD library and other parts of the software is significant. The cost of linearization and assembly of linear systems can therefore completely overwhelm the runtime of the linear solver for small problems. This computational overhead should in principle, and will also in many cases, gradually disappear with larger problem sizes. However, this is not always the case.

The first factor that may contribute to maintain a significant overhead also for larger models, is our handling of wells. Wells are generally be perforated in multiple cells throughout the grid, and to assemble their models, we need to access and extract small subsets of large sparse matrices. Such access, which some refer to as *slicing*, is not as efficient in MATLAB as one could hope, particularly not in older versions of the software. To be able to handle different types of well models like the SPE 1 case with inflow valves discussed in Section 12.4.3, the `FacilityModel` class requires a somewhat heterogeneous memory access, which we have not yet figured out how to implement efficiently in MATLAB. Using this class may therefore introduce significant overhead. For cases where all wells assume vertical equilibrium (and a Peaceman-type inflow performance relationship), you can instead use a *special facility class* called `UniformFacilityModel` that has been optimized to utilize uniform memory access. Once you have created a reservoir model, you can convert the well description to this type by the following call:

```
model.FacilityModel = UniformFacilityModel(model);
```

For the SPE 9 case discussed in Section 12.4.4, the 26 wells perforated 80 out of the 9,000 cells in the grid. Adding this transformation led to a 25% reduction in runtime on my computer.

A second factor comes from the way MRST's vectorized AD library is constructed. By default, this library always constructs a sparse matrix to represent the derivatives (Jacobian) of a vector unknown. Constructing and multiplying such matrices has a significant cost that should be avoided whenever possible. For cases with many independent primary variables, many of the Jacobian sub-matrices tend to be be diagonal. You can represent a set of diagonal matrices as a smaller, dense matrix, for which elementary operations are highly efficient. At the time of writing, work is in progress to develop more *efficient ADI backends* that utilize diagonal structures (and other known matrix patterns) to reduce construction, multiplication, and assembly costs. You can instrument you models by a specific AD back end through an assignment of the form:

```
model.AutoDiffBackend = DiagonalAutoDiffBackend;
```

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press

I expect that improved support for more efficient backends will continue to develop in future versions of the software.

Last, but not least, *plotting and creating new axes* is slow in MATLAB and has in my experience become even slower after the new graphics engine ("HG2") was introduced in R2014b. Many of the examples presented in the book have plotting integrated into simulation loops. For brevity and readability I have not optimized this part, and instead explicitly cleared axes and replotted grids and cell values whenever the plot needs to be updated. To reduce runtime, you should instead delete and recreate individual graphics handles, or simply manipulate the color data of existing graphics objects, rather than drawing the whole scene anew.

https://doi.org/10.1017/9781108591416.016 Published online by Cambridge University Press