5

Better AD Simulators with Flexible State Functions and Accurate Discretizations

OLAV MØYNER

Abstract

The `ad-core` module in the MATLAB Reservoir Simulation Toolbox (MRST) offers an object-oriented framework for rapid prototyping of new reservoir simulators based on automatic differentiation (AD-OO). The framework simplifies the task of changing and extending existing simulation models in MRST or implementing brand new ones. The MRST textbook presents a model hierarchy for the black-oil equations, discretized by a standard fully implicit method, and describes how to (automatically) select time steps and configure linear and nonlinear solvers. Herein, we present a further modularization that aims to simplify the implementation of more complex flow models and other types of discretizations and solution strategies. To this end, we view the reservoir simulator as a graph of functional relationships and their dependencies and introduce the new concept of so-called *state functions* to define these functional relationships and compute discrete quantities required for the linearized governing equations. Using the graph perspective, it is relatively simple to not only visualize and understand the data flow of highly complex reservoir simulators but also replace components of the graph and/or extend the graph with new branches as needed. The result is a versatile family of reservoir simulators that can easily be configured to run different types of multiphase, multicomponent models and at the same time support a number of different spatial and temporal discretizations. The state function concept also has a built-in compute cache that helps you to systematically eliminate redundant function evaluations. The chapter explains the new concept in detail and exemplifies its use by showcasing implicit, explicit, and adaptive-implicit simulators for the same physical processes. We also demonstrate the use of consistent and high-resolution schemes to improve simulation accuracy. Applications to complex flow physics (enhanced oil recovery, compositional flow, fractured reservoirs) are discussed in other chapters.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

5.1 Introduction

How to best organize simulators based on automatic differentiation to support efficient solution of complex multiphysics problems is still a subject of research (see, e.g., [8] for work on coupling strategies in Stanford University's research simulator, AD-GPRS). Until recently, the MATLAB Reservoir Simulation Toolbox (MRST) used a rather monolithic approach in which the implementation of reservoir and facility models was tightly coupled. In MRST 2019a, however, we decided to increase the granularity of the object-oriented, automatic differentiation (AD-OO) framework to simplify the task of adding new model equations or discretization schemes by introducing a new family of class objects, called *state functions* and state-function groups, for evaluating fluid properties, discretized fluxes, and other key properties that vary during the simulation.

This chapter discusses several of these recent additions to the AD-OO simulator framework in MRST. In particular, we will go into more detail on how the simulator performs linearizations and brings a state toward convergence. You will see how MRST recently has been extended to decouple the choice of primary variables from the governing equations to make coupling easier for multiphysics problems and to calculate sensitivities and gradients. As part of this development process, we also introduced a number of improvements to the generic model classes that enable more fine-grained control over primary variables and the strategies a model uses to update physical states so that these comply with the pertinent mathematical models. We suggest that you go through the material in chapter 12 of the MRST textbook [3] before reading this chapter.

The central concept in this chapter is the state functions that MRST uses to treat the governing equations of any model as a graph of functions that depend on each other. Taking a cue from descriptions of thermodynamics, a state function computes values that completely depend on the current state of the system, irrespective of the path taken to arrive at those values. The framework for state functions is a recent addition to MRST that addresses the growing complexity of our simulators based on automatic differentiation. It is built from the ground up to support spatial variations in functional relationships (e.g., fluid regions), variable number of phases and components (e.g., compositional or multiphase flow), and easy redefinition of discrete equations (e.g., alternative discretizations or solution strategies). You will learn how to visualize sets of state functions with complex interdependent relationships as graphs, how to modify and replace existing functions, and how to create entirely new groups of functional relationships.

Once you are familiar with how functional relationships are described with state functions, we explain how you can easily change the temporal and spatial discretizations as needed. This includes explicit and adaptive-implicit solvers, consistent discretizations for complex grids, and high-resolution schemes for transport.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

In doing so, we present the first set of MRST solvers that support both highly advanced physical effects *and* specialized discretization schemes offering improved accuracy. Altogether, the techniques in this chapter represent the next generation of the AD-OO framework, demonstrating new levels of flexibility and modularity for MRST as a prototyping platform. Further improvements to AD-OO are presented in Chapter 6, in which we describe how you can increase simulation efficiency using high-performing backends for automatic differentiation, faster linear solvers, and management of simulation cases. We recommend that you start with the chapter you are currently reading. In addition, Chapter 7 on water-based enhanced oil recovery, Chapter 8 on compositional simulation, and Chapter 11 on unified modeling of fractured media make extensive use of the new state-functions framework, demonstrating how it is used in practice.

5.2 Numerical Models in MRST

The model concept is central to MRST's AD-OO simulator framework. Whereas a mathematical model describes a system using mathematical concepts and language, a numerical model in AD-OO describes the system in a *discrete* sense. A model class, derived from the `PhysicalModel` base class (see the MRST textbook [3, section 12.1]) therefore consists of a number of different entities that together define the discrete version of pertinent physical or empirical laws. In addition, the model provides the means to modify a given discrete state of the system so that it fulfills these laws. Herein, we consider a generic system of discrete flow equations, which is described in the next two subsections.

The rest of the section then continues with an in-depth discussion of the iterative procedure models use to update states. The discussion complements chapter 12 in the MRST textbook by providing more details and outlining features that were not available in the software at the time that book was written. An updated description of the interfaces that govern nonlinear iterations is useful for readers who wish to write (or understand) advanced simulators that vary the choice of primary variables or couple different models together. In addition, some of this material is essential to motivate the state functions described in Section 5.3.

5.2.1 A Generic Multicomponent Flow Model

We consider a generic set of flow equations for a system of N individual components that have been discretized in time by a finite-difference method and in space by a finite-volume scheme over a general unstructured grid (see Figure 5.1):

$$\frac{\mathbf{M}_i^{n+1} - \mathbf{M}_i^n}{\Delta t^n} + \mathtt{div}(\mathbf{V}_i) - \mathbf{Q}_i = 0, \quad i \in \{1, \ldots, N\}. \tag{5.1}$$

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

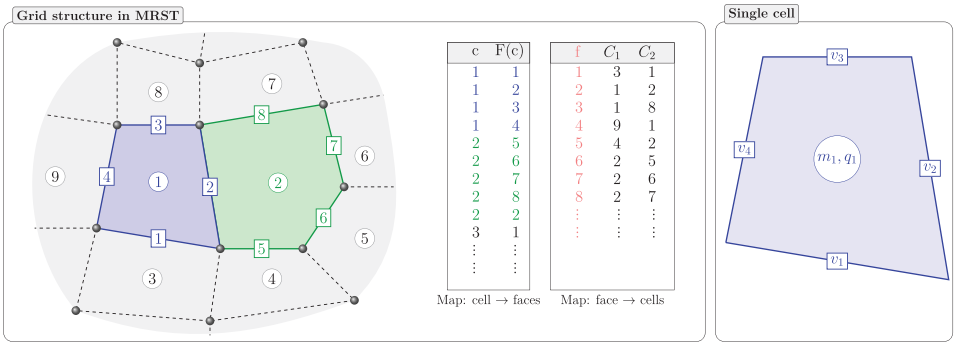


Figure 5.1 Illustration of the grid structure in MRST (left) and a finite-volume discretization on a single cell. The two tables show the mappings $c \rightarrow F(c)$ from cells to faces and $f \rightarrow \{C_1, C_2\}$ from faces to neighboring cells, used to define discrete differentiation operators. In the finite-volume discretization, we have a mass m and a volumetric source term q associated with each cell and a volumetric flux v associated with each face. These are collected to vectors $\mathbf{M}$, $\mathbf{Q}$, and $\mathbf{V}$ defined over the whole grid. In a multicomponent setting, we have one such vector for each component i.

Here, $\mathbf{M}_i$ is a vector with one entry $m_{i,\ell}$ per cell that contains the total mass of component i in cell ℓ. Likewise, $\mathbf{V}_i$ contains one value per face and corresponds to the total component mass flux integrated over that face, whereas $\mathbf{Q}_i$ is the total component mass source term in each cell. We also continue to use the notation of discrete operators from the MRST textbook [3], so that `div` is the discrete divergence operator that takes interior face values as input and produces cell values as output. We also abuse notation so that the product of two vector quantities of the same dimensions is shorthand for the Hadamard product; i.e., $(\mathbf{xy})_\ell = x_\ell\, y_\ell$, equal to the `.*` operator from MATLAB/Octave. We have intentionally left the time level at which the fluxes and source terms in (5.1) are evaluated unspecified, because there are several possibly choices, some of which are outlined in Subsection 5.4.3.

Throughout the chapter, we use Greek subscripts for phases and Latin letters for components. Each component can exist in multiple phases so that the total mass in each cell and the total component mass fluxes are both defined by a sum over all phases in which the component is present:

$$\mathbf{M}_i = \sum_\alpha \mathbf{M}_{i,\alpha} = \boldsymbol{\Phi} \sum_\alpha \boldsymbol{\rho}_{i,\alpha} \mathbf{S}_\alpha, \quad \mathbf{V}_i = \sum_\alpha \mathbf{V}_{i,\alpha}. \tag{5.2}$$

Here, $\boldsymbol{\Phi}$ is the pore volume, $\boldsymbol{\rho}_{i,\alpha}$ denotes the mass density of component i in phase α, and $\mathbf{S}_\alpha$ is the phase saturation. The convention in MRST is that, unless otherwise noted, phase quantities are given as volumes under local conditions and component quantities are given as masses.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

If we consider a multiphase extension of Darcy's law, the volumetric phase flux $\mathbf{V}_\alpha$ and component mass flux $\mathbf{V}_{i,\alpha}$ are both written via the phase-potential difference $\boldsymbol{\Theta}_\alpha$ and the face transmissibility $\mathbf{T}_f$,

$$\mathbf{V}_\alpha = -\boldsymbol{\lambda}_\alpha^f \mathbf{T}_f \boldsymbol{\Theta}_\alpha, \quad \mathbf{V}_{i,\alpha} = -\boldsymbol{\lambda}_{i,\alpha}^f \mathbf{T}_f \boldsymbol{\Theta}_\alpha. \tag{5.3}$$

The phase-potential difference is taken to be the discrete gradient of the phase pressure together with the difference in hydrostatic head over each pair of faces,

$$\boldsymbol{\Theta}_\alpha = \texttt{grad}(\mathbf{p}_\alpha) - g\,\texttt{favg}(\boldsymbol{\rho}_\alpha)\texttt{grad}(\mathbf{z}). \tag{5.4}$$

Here, $\mathbf{z}$ refers to the vector of cell center depths, g the acceleration constant, and the phase pressure $\mathbf{p}_\alpha$ is defined from some chosen reference pressure $\mathbf{p}$ by way of the capillary pressure

$$\mathbf{p}_\alpha = \mathbf{p} - \mathbf{p}_{c\alpha}. \tag{5.5}$$

We need both the *phase mobility* $\boldsymbol{\lambda}_\alpha$, which is the mobility of the *volume* of phase α in each cell, and the *component mobility* $\boldsymbol{\lambda}_{i,\alpha}$, which represents the mobility of the *mass* of component i present in phase α. One possible definition that covers many relevant models is that the phase mobility is the ratio between relative permeability $\mathbf{k}_\alpha$ and viscosity $\boldsymbol{\mu}_\alpha$. The component mobility is then equal to the phase mobility weighted by the phase density $\boldsymbol{\rho}_{i,\alpha}$:

$$\boldsymbol{\lambda}_\alpha = \frac{\mathbf{k}_\alpha}{\boldsymbol{\mu}_\alpha}, \quad \boldsymbol{\lambda}_{i,\alpha} = \boldsymbol{\rho}_{i,\alpha}\boldsymbol{\lambda}_\alpha. \tag{5.6}$$

To find the values of the mobility on each cell face, we typically employ a standard upwinding scheme. The convention for upwinding a quantity flowing with a phase α is based on the sign of the phase-potential gradient across a face connecting a pair of cells $C_1(f), C_2(f)$, defined such that a positive flux goes from C_1 to C_2,

$$\boldsymbol{\lambda}_\alpha^f = \texttt{upw}(\boldsymbol{\lambda}_\alpha), \quad \texttt{upw}(\mathbf{x})[f] = \begin{cases} \mathbf{x}_\alpha[C_1(f)], & \text{if } \boldsymbol{\Theta}_\alpha[f] \leq 0, \\ \mathbf{x}_\alpha[C_2(f)], & \text{otherwise.} \end{cases} \tag{5.7}$$

For further details on the mathematical properties of the upwinding scheme; see p. 141 of the MRST textbook [3].

The generalized set of flow equations just outlined is a useful high-level description that covers all of the different models for multiphase flow in MRST. We do not describe specific cases in further detail, because these are described in other chapters of this book or in part III of the MRST textbook [3]. Later in this chapter, we show how immiscible, black-oil, and fully compositional systems all can be posed on the form of (5.1) by choosing the appropriate definitions of $\mathbf{M}_i, \mathbf{V}_i$, and $\mathbf{Q}_i$. The definitions of these terms make up a large part of the practical implementation when extending a simulator with new flow physics. However, many of

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

the relationships we have introduced appear in (almost) the same form in several models. Implementing these as state functions makes more of the functionality in MRST *generic* and independent of the specific physical process a model may be designed for.

5.2.2 Anatomy of a `stepFunction`

As we have just seen, flow models usually constitute nonlinear systems of discrete equations that must be solved using some iterative method to correctly modify the state of the system. To this end, the default choice in MRST is to use Newton–Raphson's method, and the main workhorse is the `stepFunction` member function, which evaluates a linearized version of the physical laws, solves the resulting linear system of equations to determine an increment, and uses this increment to update the state of the system within the physical constraints the model permits. In other words, this function is responsible for defining the problem under consideration, solving it, and then defining convergence. As a consequence, it is by far the most computationally intensive part of a simulator. It is also likely the most important function for anyone who wants to develop new solvers with AD-OO.

Figure 5.2 breaks the default function down into four layers. Depending on what kind of solver you want to implement, you may interact with one or more of these layers. The outer layer of the `stepFunction` itself is well described in the MRST textbook and is shown in red. The default outer layer is in essence Newton's method applied to the system of equations. One function that may be unfamiliar from the MRST textbook is the recently introduced `reduceState` function, which removes AD variables and state-function containers from the `state` – more on that later. The `stepFunction` itself should only be replaced if you are writing a solver that does not directly use Newton's method. For instance, the sequential solution procedure implemented in the `SequentialPressureTransportModel` class from the `sequential` module has a custom implementation that amounts to two separate nonlinear solves (one solution of the pressure equation and one solve of the transport equations) inside each outer loop.

`getEquations`: Residual Equations and Canonical Primary Variables

The `getEquations` function should be well known to any existing user of AD-OO. It takes the current and previous states, together with driving forces and the corresponding timestep, and produces the linearized residual equations of the model, differentiated with respect to the *canonical set of primary variables* (unless the optional `'resOnly'` argument is enabled to only compute the residuals). The canonical primary variables should be uniquely determined from the values in the state and are the solution variables for a linearized problem.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

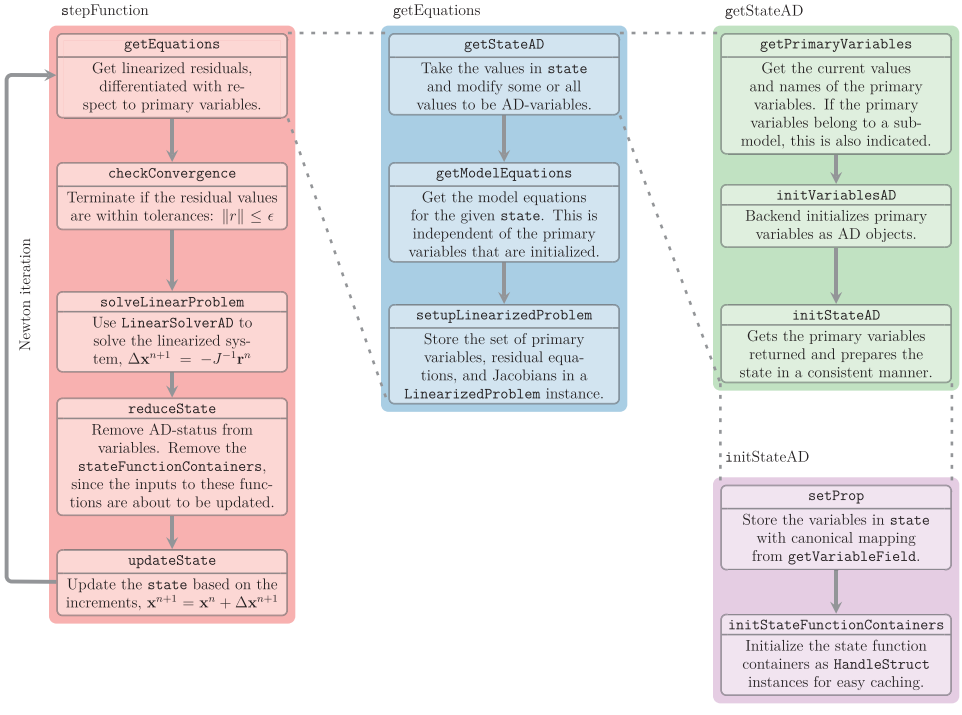


Figure 5.2 The anatomy of a `stepFunction` for a model instance. The overall default Newton logic (in red) is decomposed into a linearization of the residual equations (blue) with respect to the model's canonical primary variables (green), which are stored in a `state` (purple). Section 6.2 gives more details about the automatic differentiation libraries that perform computations behind the curtain during the linearization, and Section 6.3 discusses efficient methods for solving the linearized systems.

Historically, each MRST model has had a separate implementation of this function that sets up the canonical primary variables from the state and assembles the residual equations. Per MRST 2019a, the `PhysicalModel` base class has a default implementation that may be used:

```
if opt.reverseMode
    [state0, primaryVars] = model.getReverseStateAD(state0);
    state = model.getStateAD(state, false);
else
    [state, primaryVars] = model.getStateAD(state, ~opt.resOnly);
end
[eqs, names, types, state] = model.getModelEquations(state0, state, dt, forces);
problem = model.setupLinearizedProblem(eqs, types, names, primaryVars, state, dt);
```

We see that the forward-mode or reverse-mode AD states are initialized in separate routines before calling the new `getModelEquations` function that

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

assembles the residual equations based on the current state. This function is agnostic to the choice of primary variables. We will go into more details on the state in the next subsection. A model's choice of primary variables may vary with respect to the state; for instance, if additional variables are required to represent a gas phase that only appears at certain pressures. However, the total number of equations should normally match the total number of primary variables to give a square linear system. The equations will then be stored in a `LinearizedProblem` class instance, which can be solved with the linear solver attached to the nonlinear solver.

All model classes written prior to the 2019a release overload `getEquations` and have the entire equations setup in a single function, resulting in a fairly self-contained implementation; see, for example, `equationsBlackOil`, which implements the fully implicit discretization of the three-phase black-oil equations. There are a few drawbacks to this approach: There is inevitably some code duplication between similar models. It may also be difficult to differentiate the governing equations with respect to other variables than those in the canonical set, especially when computing sensitivities with the adjoint method or when implementing multiphysics problems for which the flow equations must be differentiated with respect to any additional primary variables required to solve for geomechanics or for thermal energy. As an alternative, it is possible to instead implement the `getModelEquations` interface that computes the residual equations directly from `state` without any awareness of the global set of primary variables. For example, the implementation used by the `GenericBlackOil` model is fairly short:

```
function [eqs,names,types,state] = getModelEquations(model,state0,state,dt,forces)
    fd = model.FlowDiscretization;
    f  = model.FacilityModel;
    [eqs, flux, names, types] = ...   % to save memory: eqs start as acc
        fd.componentConservationEquations(model, state, state0, dt);
    src = f.getComponentSources(state);
    eqs = model.insertSources(eqs, src);
    % Assemble equations and add in sources
    for i = 1:numel(eqs)
        eqs{i} = model.operators.AccDiv(eqs{i}, flux{i});
    end
    % Extend system with control equations for wells/facilities
    [we, wnm, wtyp, state] = f.getModelEquations(state0, state, dt, forces);
    eqs = [eqs, we]; names = [names, wnm]; types = [types, wtyp];
end
```

The code excerpt is somewhat simplified to only show the branch the code takes when flow is driven by wells only. The input arguments are the same as for

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

`getEquations` and any AD variables are implicitly passed on with `state`. The `FlowDiscretization` class does most of the heavy lifting to evaluate the accumulation and flux terms. This class is an example of a *state-function group* that we detail in Section 5.3. We also perform another call to `getModelEquations` – this time to the version from the `FacilityModel` – to obtain the closure relations for wells. In this way, multiple models, each with their own governing equations, can be nested or combined. Note also that we use the new `AccDiv` operator to assemble discrete equations from cell and face values. The operator implements a common compound expression and is called *after* source terms are added for efficiency. As an example of the action of this operator, we can rewrite (5.1) slightly:

$$\texttt{AccDiv}\left(\frac{\mathbf{M}_i^{n+1} - \mathbf{M}_i^n}{\Delta t^n} - \mathbf{Q}_i, \mathbf{V}_i\right) = \frac{\mathbf{M}_i^{n+1} - \mathbf{M}_i^n}{\Delta t^n} + \texttt{div}(\mathbf{V}_i) - \mathbf{Q}_i. \tag{5.8}$$

`getStateAD` and `initStateAD`

As we saw in the previous subsection, we must choose the primary unknowns the flow equations are linearized with respect to before calling `getModelEquations` to compute these equations in residual form. These unknowns must be represented as AD variables and be present in the `state` itself. Recently, we introduced the class function `getPrimaryVariables`, which is called from `getStateAD` and defines the canonical set of primary variables for a given `state`:

```
[vars, names, origin] = model.getPrimaryVariables(state);
```

The outputs are all cell arrays of equal length: `vars` contains the values of the primary variables, `names` holds their names, and `origin` is the name of the class providing the primary variables. To see how this looks like in practice, we set a breakpoint at the last line of this function (inside the `ThreePhaseBlackOilModel` class) when simulating the SPE 1 model from [7] using the `blackoil ExampleSPE1.m` script (see the MRST textbook [3, subsection 11.8.1]) and print each of the primary variables:

```
for i = 1:numel(names);
    fprintf('%8s: %3d entries from %s\n', names{i}, numel(vars{i}), origin{i});
end
```

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

```
pressure: 300 entries from GenericBlackOilModel
      sW: 300 entries from GenericBlackOilModel
       x: 300 entries from GenericBlackOilModel
     qWs:   2 entries from ExtendedFacilityModel
     qOs:   2 entries from ExtendedFacilityModel
     qGs:   2 entries from ExtendedFacilityModel
     bhp:   2 entries from ExtendedFacilityModel
```

The model contains three components, 300 cells, two wells, and three phases. The first three variables belong to the black-oil model and have values given for each cell in the domain. (Strictly speaking, it is not required that primary variables are defined in the entire model domain, but this is typical.) The remaining primary variables correspond to the facility model attached to the reservoir model. Once the simulator requested the primary variables from reservoir model, it called `getPrimaryVariables` for the facility, which returned the three phase rates and the bottom-hole pressure for each active well.

Referring back to the green box in Figure 5.2, the model can initialize any of these primary variables with the `initVariablesAD` function from the AD backend (see Chapter 6) and return the primary variables to the model through `initStateAD`, the function that makes sure the state is suitable for assembly of the discrete residual equations. The default implementation simply calls `setProp` for the primary variables. For some models, the mapping from primary variables to the state variables is not trivial and a custom function is used. One example is the black-oil equations, in which the primary variables change depending upon which phases are present; see the MRST textbook [3, subsection 11.6.4] for more details.

For the SPE 1 model just shown, some logic is required to convert the reservoir primary variables defined in every cell (`p`, `sW`, `x`) to the full set of variables (`p`, `sW`, `sO`, `sG`, `rs`) that together determine the discrete governing equations for black oil when dissolution of gas is present. In general, one must be careful to differentiate between the canonical primary variables, which may be one valid choice of many, and the variables that determine the governing equations, which are specific to a given set of equations. We could easily have modified the model to solve for primary variables (`p`, `sO`, `x`), but these primary variables would still have to be converted to (`p`, `sW`, `sO`, `sG`, `rs`) for the linearization to be successful.

Inputs to `initStateAD` are exactly the outputs from `getPrimaryVariables`, but the entries may now be of `ADI` type. The function then places the variables in the state. If the primary variables can be set directly by `setProp`, the default implementation of `initStateAD` does this automatically. Following the flow of the purple box in Figure 5.2, we see that once the `state` has been assigned all required variables, the storage for the state-function output is set up as described in Section 5.3.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

5.2.3 Validation and Preparation

When we implement a `stepFunction`, we would prefer to not have to explicitly check that the reservoir state is consistent and that the configuration of the model and boundary conditions is valid. It is better if inputs to the step function are automatically *normalized* by a set of validation routines so that all required fields are present and values are all of the expected dimensions and within expected bounds. In this way, there is no need to check for missing or ill-defined data during the linearization. Errors or warnings due to issues with user input should be provided as early as possible to avoid wasting time on simulations that will never give meaningful results. Model classes have a number of routines that perform basic validation before the main simulation starts. We quickly recap the validation routines used by AD-OO; see pp. 425 and 430 in the MRST textbook [3] for more details on the validation of model and state.

- `validateState`: Verify that the initial state completely determines the system. The model can add fields to `state` in this process. For instance, the compositional solvers discussed in Chapter 8 will perform a vapor–liquid equilibrium calculation from the initial compositions if phase mole fractions are not specified.
- `validateModel`: Prepare the model for simulation. In addition to performing various checks on static quantities not already verified in the constructor, this call will instantiate all state-function groups that belong to the model by calling the `setupStateFunctionGroupings` function. If you want to evaluate any state functions via `getProp` outside of a simulation, the relevant state functions must be present in the model by calling `model=model.validateModel()` first.
- `validateSchedule`: The function performs a validation of the entire schedule, which relies on `validateDrivingForces` to validate each set of controls.

There are also a few helper functions related to the transition from one completed timestep to the start of the solution process for the next. These are useful when designing models that need to respond to changing controls by performing additional setup based on the current solution quantities or models that approximate some parts of the system in a lagged manner from the previous step:

- `prepareReportstep`: Called before a new report step is started, when the driving forces may have changed. The function may modify the model itself to account for new wells that have become active, well targets that change, and so on.
- `prepareTimestep`: Performs setup specific to each timestep. This function is called before *any* timestep is solved.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

5.3 `StateFunctions`: Framework for AD Functions

If you are interested in understanding how the AD-OO framework evaluates different functions such as phase mobilities or densities, the `StateFunction` framework is a natural starting point. A state function in MRST is any function that is uniquely determined by the contents of the `state` struct alone: All state functions for a given model can be calculated from a single normalized state, often by expanding the state with the results of intermediate computed results from other state functions. In the following, we will motivate the mechanisms behind a simple call of the type often seen in MRST codes:

```
[mob, rho] = model.getProps(state, 'Mobility', 'Density'); % Named state functions
```

This call obtains the phase densities and mobilities for a given state. We will go into some detail on how these outputs are computed efficiently. You will also learn how to modify or replace the functions that are called behind the scenes by `getProps`. You can find the code for this section in `exampleStateFunctionsBook.m`. In addition, `stateFunctionTutorial.m` contains a self-contained tutorial showing many of the same concepts. If your interest is primarily in setting up and simulating different problems with `simulateScheduleAD`, this section can be safely skipped.

A typical conservation equation (5.1) contains a number of intermediate quantities that must be computed, such as the capillary pressure and the relative permeability, which may have different valid definitions. Often, we use many of the same quantities in multiple places: Capillary pressure, for instance, is used both when calculating phase properties that depend on the phase pressure and when calculating phase fluxes. When a quantity is required in multiple parts of the equation assembly, we would ideally prefer only to compute this quantity once for given state and primary variables. We can outline requirements on general *state functions* that depend on the current state of the physical system a model describes:

1. Dependency management: Each state function may depend on many other state functions, which in turn may have additional dependencies. Keeping track of the dependency graph of any given function can become nontrivial. Programming is simplified if the simulator has a mechanism to ensure that all input quantities to a function have been evaluated before evaluating the function itself.
2. Generic interfaces: Closure relationships and physical properties can often be modeled using different functional dependencies and different versions of the same family of mathematical (flow) models may not always have the exact same physical variables (gas may, for instance, not always be present). A numerical model requiring some physical property G should preferably therefore not explicitly define relations on the form $G(p, s_w)$.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

3. Lazy evaluation with caching: State functions are only evaluated if needed either directly or indirectly as a dependency for another function. Already evaluated values are stored and can be retrieved with minimal additional cost.
4. Support for spatial dependence: MRST uses a vectorized syntax for efficiency when evaluating constitutive relationships over all cells or faces in a grid, but a common requirement in Darcy flow applications is that such relationships vary throughout the domain to account for variations in the properties of the underlying porous media. The most typical example is rock types that are characterized by different relative permeability or capillary pressure curves, but there are also other types of relationships where a given function varies spatially. For example, in ECLIPSE input, such variations appear in the form of pressure–volume–temperature (PVT), saturation, and equilibration regions.
5. Implementations should be independent of the chosen primary variables to support calculation of sensitivities.

Starting in MRST 2019a, AD-OO simulators primarily evaluate properties and functional relationship via a state-function framework. Specific state functions are implemented as subclasses of `StateFunction`, whereas a `StateFunctionGrouping` instance groups interdependent state functions together and provides mechanisms for lazy evaluation of these functions with caching.

5.3.1 A Crash Course in State Functions

State functions were introduced in MRST as helper classes inside numerical simulation models, but they can also be used independent of a model. Before we look at the functions used to simulate multiphase flow in MRST, we discuss a simple example that demonstrates the fundamentals of state functions. Assume that we want to compute the following function using state functions:

$$G(x, y, a, b) = xy + ab. \tag{5.9}$$

The state object will in this case be a structure that contains, as a minimum, the four fields, `x`, `y`, `a`, and `b`, holding the numbers x, y, a, and b. The generic way to evaluate (5.9) will then be of the form `G(state)`. Following operator precedence, we can decompose this function into two products, xy and ab, and the addition of the two results. Implementing a state function consists of two main parts: Any functional dependencies must be documented as part of the class constructor, whereas the function is evaluated using the `evaluateOnDomain` member function. The constructor of the base class has a simple signature:

```
function sfn = StateFunction(model, regions)
```

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

The model class is normally a required input argument, because the definition of a state function depends on the grid and physical model under consideration. The second input variable is optional and can be used to specify spatial dependence. For functions such as capillary pressure that produce results in each cell of the domain, `regions` would be a vector with one entry per cell that indicates which capillary pressure curve is to be used in that cell.

We first define a state-function class, named `TutorialNumber`, that simply retrieves any of the numbers x, y, a, b, from the corresponding named field in the `state` struct. The constructor of this state function specifies dependency on the state property with name `n` (which in our case will be `'x'`, `'y'`, `'a'`, or `'b'`):

```
function tn = TutorialNumber(n)   % Class constructor
    tn.stateField = n;
    tn = tn.dependsOn(n, 'state');
end
function v = evaluateOnDomain(tn, model, state) % Main evaluator
    fprintf('Retrieving %s from state.\n', tn.stateField);
    v = state.(tn.stateField);
end
```

This is an *external* dependency, because the values are retrieved from outside of the current group of functions, in this case from the state. The evaluation simply returns the value from the state. For pedagogical purposes, the function outputs a log message when evaluated.

Next, we implement the product as another state function taking two inputs:

```
function tp = TutorialProduct(left, right)    % Class constructor
    [tp.leftNumber, tp.rightNumber] = deal(left, right);
    tp = tp.dependsOn({left, right});
end
function v = evaluateOnDomain(tp, model, state)
    [l, r] = tp.getEvaluatedDependencies(state, tp.leftNumber, tp.rightNumber);
    fprintf('Multiplying %s and %s.\n', tp.leftNumber, tp.rightNumber);
    v = l.*r; % Perform element-wise multiplication
end
```

The constructor declares *internal* dependencies on two named functions that are implied to be from the same group as the product. The evaluation function can then use the function `getEvaluatedDependencies` to obtain these values; i.e., retrieve them from cache if they have already been computed. We also write a nearly identical `TutorialPlus` that adds two numbers together, which is omitted here.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

We now have the three classes needed to implement (5.9). We instantiate an empty group that will manage the relationships between the state functions:

```
group = StateFunctionGrouping('mystorage'); % Store in state.mystorage
```

Here, the argument specifies that outputs for this group are stored in the `mystorage` field of the state. We can now define four functions that each retrieve a specific number from the state:

```
group = group.setStateFunction('A', TutorialNumber('a')); % state.a -> A
group = group.setStateFunction('B', TutorialNumber('b')); % state.b -> B
group = group.setStateFunction('X', TutorialNumber('x')); % state.x -> X
group = group.setStateFunction('Y', TutorialNumber('y')); % state.y -> Y
```

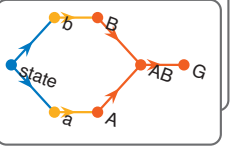
We then add the functions required to compute G and show the final class instance:

```
group = group.setStateFunction('AB', TutorialProduct('A', 'B')); % A*B
group = group.setStateFunction('XY', TutorialProduct('X', 'Y')); % X*Y
group = group.setStateFunction('G', TutorialAdd('AB', 'XY'));    % AB + XY
disp(group)
```

```
StateFunctionGrouping (edit|plot) state function grouping instance.
StateFunctionGrouping has no intrinsic state functions.

Extra state functions (Added to class instance):
A: TutorialNumber (edit|plot)
B: TutorialNumber (edit|plot)
X: TutorialNumber (edit|plot)
Y: TutorialNumber (edit|plot)
AB: TutorialProduct (edit|plot)
XY: TutorialProduct (edit|plot)
G: TutorialAdd (edit|plot)
```

```
classdef TutorialProduct < StateFunction
    properties
        leftNumber
        rightNumber
    end
    methods
```



All state-function groupings have a custom `disp` implementation that makes it easy to open the documentation by clicking the class name, open the implementation in the editor by clicking "edit," or make a plot of the functions by clicking "plot." In the listing, we have also shown the plot of the state-function group as it appears when visualized from within MATLAB. Alternatively, we can also convert the graph to a LaTeX file and compile it if TikZ is available, resulting in Figure 5.3.[1] We

[1] TikZ plots of state functions use the graph-drawing functionality [10] for automatic layouts.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

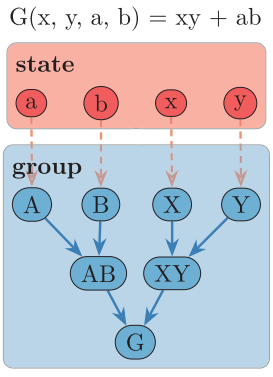


Figure 5.3 State-function graph for the crash course example. The relationships between the state variables, shown in red, and the state functions in a group, shown in blue, are shown by directed connections that indicate a dependency. For this figure, we have converted the graph to a TikZ file, which can be manually tweaked to get a publication-ready figure.

will revisit the plotting functionality later on in this chapter. Note that whereas we have seven different state functions, we only implemented three classes. The same implementation is in this case used for several different named functions. Thinking about how to structure state functions can drastically reduce the amount of code you have to write.

Visualizing `StateFunctionGrouping` instances uses the MATLAB graph-plotting functionality, which was introduced in R2015b. The plots presented in this chapter are generated as TikZ from these graphs and are somewhat different from the plots seen in the MATLAB plotting interface. The 2015b requirement does not apply to simulators that employ state functions; it is only a requirement for the graph plotting.

We are now ready to define our input. The state functions pick values from the state, so we initialize a struct with arbitrary values for the required fields:

```
state0 = struct('a', 3, 'b', 5, 'x', 7, 'y', 2); % Initial state
```

We next add the state-function *container* for the group to enable caching. (During a simulation, this would be done automatically by `initStateAD`.) Per the constructor call to the group, intermediate results will be stored under the field `mystorage` as a specialized struct that acts as a handle class:

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

```
state = group.initStateFunctionContainer(state0); % Enable caching
disp(state.mystorage)
```

```
HandleStruct with fields:
     A: []
     B: []
     X: []
     Y: []
    AB: []
    XY: []
     G: []
```

Now that all functions and inputs are set up, we can finally evaluate G, which will trigger the evaluation of all intermediate functions as the dependency graph is traversed. We insert an empty array in the positional argument normally reserved for the model, because it is not needed for our example:

```
G = group.get([], state, 'G'); % First call, triggers execution
```

```
Retrieving a from state.
Retrieving b from state.
Multiplying A and B.
Retrieving x from state.
Retrieving y from state.
Multiplying X and Y.
Adding AB and XY.
```

We see that the dependencies are traversed in a depth-first manner of the transpose of the directed graph in Figure 5.3, starting from G. The container is now updated with the intermediate results. As we just saw, storage of evaluated state functions is achieved through the use of a `handle` class instance, which acts much like a pointer in C++.[2] The evaluated properties will still be cached if you pass an initialized AD state to a function that calls `getProp` without returning the `state` itself:

```
disp(state.mystorage)
```

```
HandleStruct with fields:
     A: 3
     B: 5
     X: 7
     Y: 2
    AB: 15
    XY: 14
     G: 29
```

[2] See subsection 12.3.2, *Handle classes*, of the MRST textbook [3] for more details and the design considerations that lead to AD-OO's sparing use of handle as the base class.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

We can verify this by calling the `get` function again; no output will be produced, because the result is already computed:

```
G = group.get([], state, 'G') % Second call, cached
```

```
G =    29
```

With only a few different functions present, it is fairly simple to see from the graphs whether anything is missing or out of place. We can also use the `checkDependencies` function that checks all dependencies and optionally returns the boolean `ok` indicating whether all of the dependencies are fulfilled:

```
mrstVerbose on; % Print missing dependencies
ok = group.checkDependencies();
```

```
All internal dependencies are met for group of class
StateFunctionGrouping.
```

In sum, we can safely request all functions in this group via `getProp`. We now add a function `F` that multiplies `X` with a missing function `Z` and validate again:

```
group = group.setStateFunction('F', TutorialProduct('X', 'Z'));
ok = group.checkDependencies();
```

```
Unmet internal dependency for F in StateFunctionGrouping:
  Did not find Z in own group.
```

It is worth nothing that if no output argument is given, the function will throw an error at the first unmet dependency it finds. You can also pass in a cell array of other groups to check any external dependencies:

```
ok = group.checkDependencies({groupA, groupB, groupC});
```

The `validateModel` function discussed in Subsection 5.2.3 checks all external and internal dependencies at the start of a simulation.

5.3.2 Evaluation of Properties

In the preceding sections, we have not specified the logic behind `getProp` and how it triggers an evaluation of one or more state functions. Figure 5.4 outlines the process MRST goes through to produce outputs for the given state and a

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

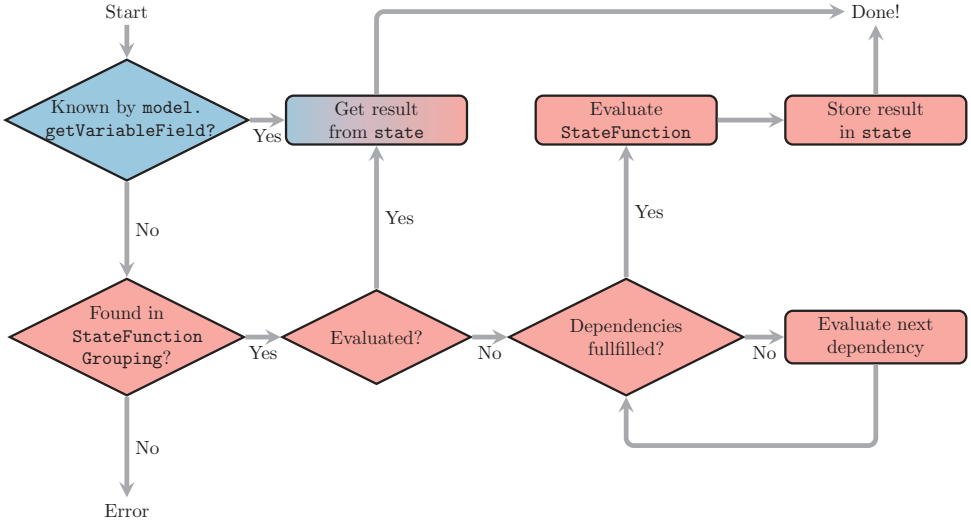


Figure 5.4 Flowchart demonstrating the internal logic used when calling `model.getProp` to get the value of some state variable or state function in the AD framework. The blue color corresponds to the model itself, whereas nodes colored in red are handled by any `StateFunctionGroupings` attached to the model.

named property. We can make a few observations from the flowchart: Named properties known by `model.getVariableField` take precedence over any named state functions present. The intent is to differentiate between the properties present in `state`, which together completely define the current conditions of the system we are simulating, and properties that can be derived from the current conditions via functions. By convention, state-function groups are instantiated at the start of the simulation by the `validateModel` function, which calls the `setupStateFunctionGroupings` function. We do not create them in the model constructor, because the properties of the model can be changed after the class is instantiated and the choice of specific functions depends on the final configuration.

We also see that the evaluation of a property is *cached* so that it will be evaluated only if not already present in `state`. If dependencies are not yet evaluated, the simulator will evaluate them in sequence as we saw earlier. Each of the dependencies may have their own dependencies that are recursively computed. For this reason, the first call to `getProp` within a linearization is often the slowest,

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

because it will implicitly calculate many values. In the initialization of the AD state in `initStateAD`, described in Subsection 5.2.2, the `initStateFunctionContainers` function will give each `StateFunctionGrouping` attached to the model the opportunity to initialize storage for all known state functions. This call is responsible for setting up the containers we saw attached to the state in Subsection 5.3.2. The storage and any Jacobians are removed from `state` using the `reduceState` function before the variables in `state` are updated and passed on to the next nonlinear iteration.

Let us compute the density for a compositional model by initializing a state with caching for a model with 40 000 cells. Here, we have performed the evaluation without the initialization of primary variables[3]:

```
fprintf('First call:  '); tic(); rho = model.getProp(state, 'Density'); toc();
fprintf('Second call: '); tic(); rho = model.getProp(state, 'Density'); toc();
```

```
First call: Elapsed time is 0.035023 seconds.
Second call: Elapsed time is 0.000847 seconds.
```

Two calls will only evaluate the state functions necessary to compute density once so that the second call just retrieves the stored result. Let us examine the relevant fields of the state-container storage after `initStateAD` has been called but before we call `getProp`:

```
disp(state.PVTProps)
```

```
                        :
                  Density: []
            PhasePressures: []
PhaseCompressibilityFactors: []
                        :
```

These empty values, corresponding to density and two other properties required to evaluate density, are filled in after calling `getProp` once:

```
                        :
                  Density: {1×3 cell}
            PhasePressures: {1×3 cell}
PhaseCompressibilityFactors: {1×3 cell}
                        :
```

[3] For more details, see Subsection 5.2.2 and then call `getProp` twice. The setup code for this snippet is found in `examplePlottingStateFunctionsBook.m`.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

As in the state-function crash course, we see that the outputs from the (intermediate) function evaluations have been stored in the state object. For this reason, once a state has been initialized with state-function caching of variables – e.g., by calling `getStateAD` – you should be careful not to modify any values of the state. Otherwise, the output of `getProp` may not be what you expect due to caching.

There are actually two main interfaces for retrieving state-function outputs. You have already been introduced to the general `getProp`/`getProps` interface, which automatically figures out whether the input arguments belong to the state itself or to one of the state-function groups. The second interface is specific to each grouping and is required if properties in multiple groups have the same name:

```
model.getProp(state, 'Mobility')   % Standard interface: Automatic mapping
fp = model.FlowPropertyFunctions;  % StateFunctionGrouping class instance
fp.get(model, state, 'Mobility')   % Internal interface: Get from grouping
```

The model can contain several different state-function groups. For a group to be accessible to `getProp`, it must be returned from the model member function `getStateFunctionGroupings`. This function outputs a cell array of all currently initialized groups. For example, if we write a custom model class inheriting from `PhysicalModel` that has a new group contained in the field `MyCustomGroup`, we must modify the member function so that it also reports this new group:

```
function groupings = getStateFunctionGroupings(model)
    groupings@PhysicalModel(model);         % Get parent groups
    groupings{end+1} = model.MyCustomGroup; % Output new group
end
```

5.3.3 Examples of State Functions

You have seen how the state functions work on a simplified example and where they fit as a part of a model. Next, we consider a few examples of specific functions used by MRST for black-oil simulation to demonstrate the usage in practice. The first example is the `PoreVolume` state function, which provides the discrete pore volume to the simulator. The base implementation inherits from the `StateFunction` base class and has a simple constructor that validates the cell-wise pore volumes stored in the model. This class is stored under `ad-core/statefunctions/flowprops`. In the constructor, we verify that the

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

model has pore volumes stored and that these are nonnegative and given for all cells in the domain. During simulation, the simulator will call the `evaluateOnDomain` member function with an AD-initialized state and the current model. In this case, we merely retrieve the pore volume from the model:

```
function gp = PoreVolume(model, varargin)
    gp@StateFunction(model, varargin{:});  % Call base constructor
    assert(isfield(model.operators, 'pv'), ...
                'Pore volume (pv) must be present as field in operators struct');
    assert(numel(model.operators.pv) == model.G.cells.num,...
                'Pore volumes must be defined in each cell.');
    assert(all(model.operators.pv > 0), 'Pore volumes must be non-negative.');
end

function pv = evaluateOnDomain(prop, model, state)
    pv = model.operators.pv; % Static pore-volume
end
```

Pore volume is often thought of as static, and if it changes it is usually through a weak dependence on the pressure as the rock surrounding the pores expands or contracts to match the stresses from changes in the fluid pressure. This alternative definition writes $\mathbf{\Phi}$ as a function of average porosity $\boldsymbol{\phi}$ in each cell, bulk volume $\mathbf{V}$ in each cell, and a pressure-dependent multiplier function $\mathbf{m}$, so that $\mathbf{\Phi}(\mathbf{p}) = \mathbf{m}(\mathbf{p})\boldsymbol{\phi}\mathbf{V}$. The `BlackOilPoreVolume` specialization inherits from `PoreVolume` and documents in the constructor that the class requires the pressure from the state to perform an evaluation. We also verify that the fluid struct contains the requisite function handle for the multiplier. The function evaluation now consists of two parts: We first retrieve the pore volume from the model by calling the base implementation and then evaluate and apply the multiplier to the volumes in each cell:

```
function gp = BlackOilPoreVolume(model, varargin)
    gp@PoreVolume(model, varargin{:});
    gp = gp.dependsOn({'pressure'}, 'state');
end
function pv = evaluateOnDomain(prop, model, state)
    % Get effective pore-volume, accounting for rock-compressibility
    pv = evaluateOnDomain@PoreVolume(prop, model, state);
    f = model.fluid;
    p = model.getProp(state, 'pressure');
    pvMult = prop.evaluateFluid(model, 'pvMultR', p);
    pv = pv.*pvMult;
end
```

Here, `pvMultR` is passed onto `evaluateFunctionOnDomainWithArguments`, a special function that evaluates one or more `function_handle` instances on the

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

entire domain with the given arguments via `evaluateFluid`. If `fluid.pvMultR` is a single function, it is interpreted as valid for all cells in the domain. Otherwise, the function is evaluated for each subdomain. The code executed by the class is then equivalent to the following:

```
f = model.fluid;
if iscell(f.pvMult)  % Regions present
    v = model.AutoDiffBackend.convertToAD(zeros(model.G.cells.num, 1), p);
    for i = 1:numel(f.pvMult)      % Iterate over regions
        active = fn.regions == i;  % Find the cells where function is valid
        v(active) = f.pvMultR{i}(p(active)) % Evaluate for these cells
    end
else                % No regions present
    v = f.pvMult(p);
end
```

The assumption of small variation can be violated; for instance, if the bulk mass of the porous medium is fractured during simulation or if the rock reacts with the fluid through mineralization or acidification. By exposing the pore volume as a state function, we have made it easy to replace the implementation for practitioners who wish to study different phenomena. Such an implementation has access to all current primary variables in `state` and all static properties in the model, making it possible to introduce any functional dependency in a simulator without modifying any of the other code. Breaking a complex simulator into many smaller constitutive parts that each represents a function with a single output makes it easier to test and verify correct behavior for the simulator as a whole. In the next section, we describe exactly how the interplay between different functions is managed by the simulator and how to interject new relationships as needed.

5.3.4 The `StateFunctionGrouping` *Class*

The typical simulator will contain a fairly large number of different functions that evaluate functional relationships based on the reservoir state. Many of these functions will depend on each other and may be closely related. The `StateFunctionGrouping` class groups many different `StateFunction` instances together and manages the evaluation of entries in the group. In the previous section, we saw how two different pore volume implementations can coexist, but we were vague on how the simulator would choose the correct one for a given scenario. The answer lies in the corresponding state-function group for PVT properties.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

We already saw an example of how to use the groups in Subsection 5.3.1. The group is instantiated during model validation and chooses the specific implementations for functions for the specific scenario. We start by calling the base constructor and retrieving the PVT regions:

```
function props = PVTPropertyFunctions(model)
    props@StateFunctionGrouping();
    pvt = props.getRegionPVT(model);
```

Each state function is then constructed with the region indicators:

```
bf = BlackOilShrinkageFactors(model, pvt);
props = props.setStateFunction('ShrinkageFactors', bf);
```

Certain properties have multiple implementations. One example is the pore volume we have already discussed:

```
if isfield(model.fluid, 'pvMultR') % Check for multiplier
    pv = BlackOilPoreVolume(model, pvt);
else
    pv = PoreVolume(model, pvt);
end
props = props.setStateFunction('PoreVolume', pv);
```

It is instructive to consider another state function from the PVT grouping. Let us consider a black-oil model, in which the gas component is allowed to dissolve into the oleic phase. The multiphase densities are then given as a function of the surface densities, the shrinkage factors b_α, and the solution gas–oil ratio R_s,

$$\rho_w = b_w \rho_{ws}, \quad \rho_o = b_o(\rho_{os} + R_s \rho_{gs}), \quad \rho_g = b_g \rho_{gs}. \tag{5.10}$$

This equation is included as an example of a function with multiple dependencies. You can consult subsection 8.2.3 and section 11.4 of the MRST textbook [3] for additional details on the black-oil model and the physical interpretation of these terms. To keep the discussion straightforward, we consider a simplified version of the general MRST class `BlackOilDensity`, in which we assume that gas can always dissolve into the oleic phase and that the oil component does not vaporize into the gaseous phase. We consider the constructor, where we have two types of dependencies documented:

```
function gp = BlackOilDensity(model, varargin)
    gp@StateFunction(model, varargin{:});
    gp = gp.dependsOn({'rs'}, 'state');
    gp = gp.dependsOn({'ShrinkageFactors'});
end
```

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

The first dependency is an *external* dependency on `rs` from `state`. The second is an *internal* dependency, in which we require values of the named state function `ShrinkageFactors` from the same *group*: We are not specifying the name of this group itself, but we are implicitly saying that if the simulator wants to evaluate `BlackOilDensity`, it would also have to provide some way of getting `ShrinkageFactors` in the same group. Internal dependencies are always evaluated with the same state object and can be thought of as closely related. The interface for evaluating the properties reflects this division between internal and external dependencies: For each internal dependency, we can use `getEvaluatedDependencies` to query whether this property already has been evaluated (and cached) so that we do not have to use a potentially expensive call to `getProp`. This function bypasses the more expensive parts of `getProp` that check and evaluate dependencies. Summing up, the evaluation of black-oil densities is implemented as follows:

```
function rho = evaluateOnDomain(prop, model, state)
    rs = model.getProp(state, 'rs');  % External dependency
    b = prop.getEvaluatedDependencies(state, 'ShrinkageFactors'); % Internal
    rhoS = model.getSurfaceDensities(); nph = model.getNumberOfPhases();
    rho = cell(1, nph);  % Allocate storage
    for i = 1:nph
        rho{i} = rhoS(prop.regions, i).*b{i}; % rhoS_alpha * b_alpha
    end
    oix = model.getPhaseIndex('O'); gix = model.getPhaseIndex('G');
    rho{oix} = rho{oix} + rs.*b{oix}.*rhoS(prop.regions, gix);  % Disgas
end
```

The member function `evaluateOnDomain` is the main way of getting values from state functions. It can generally be implemented just as you would implement any other function, with a few caveats:

- Matrix output is generally not supported when a function should support automatic differentiation in MRST. Unless your function will never produce AD outputs, consider using a cell array of column vectors instead, where each entry corresponds to one column of the matrix.
- If the output is made up of either a single column vector or multiple column vectors arranged as a row-cell array, the values should match the dimensions of `prop.regions` if present.
- If a cell array is given as output, the convention is to let the row index correspond to the index of a component and the column index indicate the phase. It is assumed that calling `value` on the output is safe, which requires that all vectors in the same row of a cell array are of the same length.
- Empty entries in cell arrays will be interpreted as zero values.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

Shrinkage factors are listed as *internal* dependencies because they belong to the `PVTPropertyFunctions` grouping that contains all properties related to modeling PVT behavior of fluids. When a reservoir model is validated before simulation, the default `PVTPropertyFunctions` class will be instantiated, and specialized versions of the different state functions will be chosen based on features present in the model. For black-oil models, this includes the `disgas` and `vapoil` flags, the function handles present in `state.fluid`, and an Eclipse deck, if present, in `model.inputdata`. To illustrate, we examine the flow functions for the SPE 1 model from `blackoilTutorialSPE1`:

```
model = model.setupStateFunctionGroupings(); % Setup
disp(model.PVTPropertyFunctions)
```

```
PVTPropertyFunctions (edit|plot) state function grouping instance.

Intrinsic state functions (Class properties for PVTPropertyFunctions,
always present):
                   Density: BlackOilDensity (edit|plot)
            PhasePressures: PhasePressures (edit|plot)
                PoreVolume: BlackOilPoreVolume (edit|plot)
 PressureReductionFactors: BlackOilPressureReductionFactors (edit|plot)
          ShrinkageFactors: BlackOilShrinkageFactors (edit|plot)
                 Viscosity: BlackOilViscosity (edit|plot)

Extra state functions (Added to class instance):
                     RsMax: RsMax (edit|plot)
```

The output, produced by the `StateFunctionGrouping` base class's overloaded implementation of `disp`, shows a number of *intrinsic* PVT properties that will always be present in a black-oil model, such as density and pore volumes discussed earlier in this chapter, as well as phase pressures, shrinkage factors, and viscosities. The pressure reduction factors are weights that can be used to transform a set of conservation equations into a pressure equation (cf. the discussion of constrained pressure residuals in subsection 12.3.4 of the MRST textbook [3]). When writing a general function that acts on some model with a PVT property-functions instance, we can safely assume that the outputs of these state functions are available through `model.getProp`. In addition, `RsMax` has been added as an extra property to facilitate support for variable bubble-point pressure. We cannot generally assume that this function is available, because it implicitly relies on the specific way dissolution is modeled in black-oil-type equations. We also see that each property, intrinsic or not, has both a group name (e.g., `Density`) and an implementation name for the configured class (`BlackOilDensity`). We can look up the documentation or edit the specific implementation directly from the command window or plot the

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

relationship between the functions. We can easily add, get a handle to, replace, remove, and evaluate functions for a given state-function grouping `g` through the public interfaces:

```
g = g.setStateFunction('Name', impl);   % Set 'Name' implementation
p = props.getStateFunction('Name');     % Get class instance defining 'Name'
g = g.removeStateFunction('Name');      % Only allowed for extra properties
v = props.get(model, state, 'Density'); % Evaluate for given state
```

By adding or changing which `StateFunction` class is used to evaluate a given physical property, it is possible to effectively rewrite or reconfigure many parts of the simulator without changing other parts of the existing code. Once a model has been validated to set up the state-function groups, we can compute values for the named property for a given `state` through the standard interface:

```
v = model.getProp(state, 'Density'); % Standard interface
```

Generally, you do not need to know in which group a specific state function is placed. However, if two different properties from different groups have the same name, you should use the internal interface from Subsection 5.3.2.

5.4 Discretization with State Functions

In a flow simulator, the `PVTPropertyFunctions` grouping is usually accompanied by other groups like the `FlowPropertyFunctions`, which contains flow properties such as capillary pressure, relative permeabilities, mobilities, etc. Other groupings include `FlowDiscretization`, which holds functions used to discretize and define the mass-conservation equations for a typical finite-volume scheme, and `FacilityFlowDiscretization`, which contains functions needed to compute fluxes from wells for a given set of controls. Neither of these groups of functions is entirely independent of each other, and we can thus also have a number of cross-group dependencies. In summary, state functions are used to evaluate properties but are also used to discretize model equations. In this section, we will go through some of the benefits of using state functions to build a simulator.

Much of the functionality in this section is only supported in the `Generic` family of AD models. Model classes that explicitly name the number of phases – e.g., `ThreePhaseBlackOilModel` – are currently limited to fully implicit schemes with the standard spatial discretizations.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

5.4.1 The Simulator as a Graph

There may be many state functions involved in a simulator; at the time of writing, the solvers in the `compositional` module define nearly 40 individual functions when wells are present. We have seen that there are numerous benefits associated with this structure, but it may become unwieldy to manage such a large number of classes if you are not already intimately familiar with the structure of the simulator. One way to visualize this structure is to consider the dependency graph as we briefly saw in Subsection 5.3.1: If you think of each state function as a node and treat the dependencies of each functions as edges in a graph, each property evaluation is represented as the traversal of a directed graph. Let us return to the mass densities from (5.13). If a model has a density implementation, we can easily plot the function and all of its dependencies by calling[4]:

```
plotStateFunctionGroupings(model, 'Stop', 'Density')
```

The optional `'Stop'` argument indicates that we only wish to plot the part of the graph visited by a backwards traversal starting from the density. This gives us a plot of all functions and state variables needed to evaluate the density. The dependency graph in Figure 5.5 shows that the density in a black-oil model depends directly on b-factors and R_s, as expected from (5.13). It also depends indirectly on saturation through capillary pressure, which enters the phase pressures used to evaluate the b-factors. The black-oil model represents a simplified PVT behavior compared to a compositional model. Figure 5.6 shows the result of the same plotting command applied to a compositional model. Compositional models compute density directly, and the b-factors (`ShrinkageFactors`) are thus nowhere to be seen. Density of the liquid phase, for example, depends on pressure, temperature, phase compressibility factor Z_ℓ, and phase mole fractions x_i,

$$pV = n_\ell RTZ_\ell \rightarrow \rho_\ell = \frac{p}{RTZ_\ell}\sum_{i=1}^{N_c} M_i x_i.$$

You can use standard techniques to manipulate the graphs, and it is possible to quickly get an overview of complex simulators simply by plotting one or more of the function groups in the simulator. Plotting the full graph is done by either passing the model to plot all groups or the desired groups as a cell array:

```
plotStateFunctionGroupings(model)
plotStateFunctionGroupings(model.getStateFunctionGroupings())
```

[4] State functions are set up at the start of a simulation, with reasonable defaults given. If you have just constructed your model (e.g., using `initEclipseProblemAD`), you must issue a call to `model = model.validateModel()` to set up the state functions and invoke the defaults.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

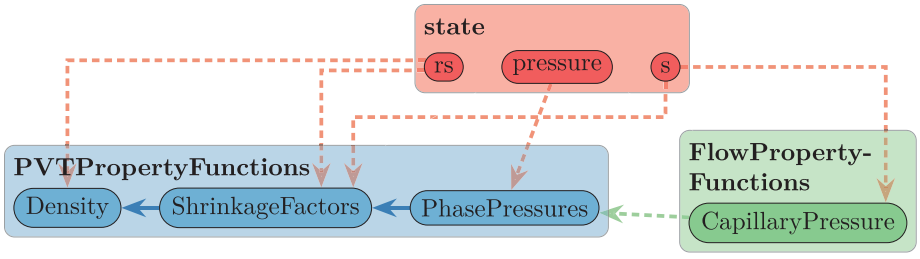


Figure 5.5 Dependency graph for phase mass density in a standard black-oil model with gas dissolved in oil.

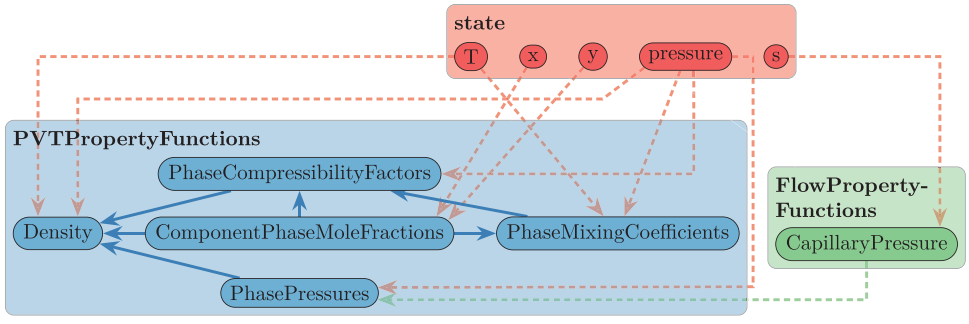


Figure 5.6 Dependency graph for phase mass density in a three-phase compositional model; PVT behavior is clearly more complex than for the black-oil model in Figure 5.5.

Graphs with a large number of connections are often difficult to navigate but may serve as a good starting point for exploring the interdependencies of different functions used in a given simulator.

The optional argument `'label'` determines the type of labels used for the nodes in the graph. The default is `'name'`, which labels nodes by the name of the corresponding state function has been given in the group. If we instead use the value `'label'`, each state function outputs the contents of its `label` property instead, which may be interpreted by TEX. As an example, Figure 5.7 reports the entire graph of the `GenericBlackOilModel` for SPE 1, including `state` variables, flow properties, flux discretization for the reservoir equations, and facility flux discretization for the wells. The default symbols used in the plot correspond to the notation from the set of generic governing equations in Subsection 5.2.1, but you can, of course, change the labels if you want to use generate figures with another notation.

Returning to (5.1), we see that all three discrete quantities we need to evaluate the mass-conservation equation for a component appear as leaf nodes in Figure 5.7:

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

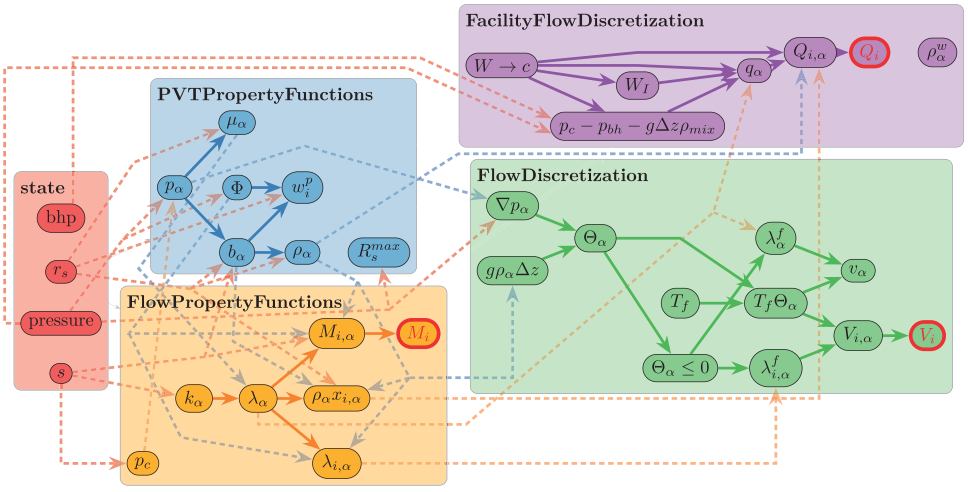


Figure 5.7 The entire state-function graph for a black-oil model. The `state` variables are the starting point for computing the reservoir flow properties, PVT properties, reservoir fluxes, and well source terms. The terms appearing in (5.1) are marked in red.

The component mass in each cell $\mathbf{M}_i$ is in the flow property functions, the component mass mflux $\mathbf{V}_i$ across each cell interface in the flux discretization, and the component mass source term $\mathbf{Q}_i$ is the terminus of the facility flux discretization.

5.4.2 The Component Implementation

State functions relating to compositions have so far always given values for all components simultaneously. This is manageable for many simple fluid descriptions but can cause code duplication for multicomponent systems, because the same component may be present in many different settings. For example, the immiscible water component is present in both black-oil and compositional systems. Generic models therefore make use of a helper class, `GenericComponent`, that is instantiated by `validateModel` for each component present. The generic black-oil model, for example, executes the following when either dissolved gas or vaporized oil is present:

```
for ph = 1:nph
    switch names(ph)
        case 'W',   c = ImmiscibleComponent('water', ph);
        case 'O',   c = OilComponent('oil', ph, disgas, vapoil);
        case 'G',   c = GasComponent('gas', ph, disgas, vapoil);
    end
    model.Components{ph} = c;
end
```

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

The water phase is treated as a generic immiscible fluid, whereas the gas and oil components use specialized classes. We will go through these in more detail in the following subsections. The component base class implements the component phase mass from (5.2) and the component mass mobility from (5.6). In the following, we focus on the definition of the phase-component mass density $\boldsymbol{\rho}_{i,\alpha}$ through the function `getComponentDensity`, because it implicitly defines the component mobility and mass fractions. The possibility exists for overriding each of these functions individually, should you, for example, desire a more complex expression for (5.6). For additional examples of specialized components for different applications, see Chapters 7 and 8.

Immiscible Components

Immiscible models assume that a component belongs to a single phase only and that the phase is only composed of that component. Phases and components are often used interchangeably when describing models in which all phases are immiscible. As with a state function, we document the dependencies the component have in the constructor. For the immiscible component, we pass the index of the phase the component belongs to as a required parameter:

```
function c = ImmiscibleComponent(name, phase)
  c@GenericComponent(name);
  c.phaseIndex = phase;
  c=c.functionDependsOn('getComponentDensity','Density','PVTPropertyFunctions');
end
```

If we let $L(i)$ label the phase component i belongs to, we can define the mass density for the component succinctly:

$$\boldsymbol{\rho}_{i,\alpha} = \begin{cases} \boldsymbol{\rho}_{\alpha}, & \text{if } \alpha = L(i), \\ 0, & \text{otherwise.} \end{cases} \tag{5.11}$$

To define the component mass density, we let the base class produce an empty cell array and then set the component density equal to the phase density in phase $L(i)$:

```
function c = getComponentDensity(component, model, state)
    rho = model.getProp(state, 'Density');
    c{component.phaseIndex} = rho{component.phaseIndex};
end
```

In addition, there are a number of functions that describe how a component is divided between the different phase streams at surface phase conditions (i.e., as `qWs`, `qOs` and `qGs`) and how compositions are added to the flow from an injector well. We associate each unit of mass produced of the immiscible component

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

with one unit of the corresponding surface phase stream, regardless of the pressure and temperature conditions:

```
function c = getPhaseCompositionSurface(component, model, state, pressure, T)
    c = cell(model.getNumberOfPhases(), 1);
    c{component.phaseIndex} = 1;
end
```

Finally, we can define the mass fraction for the injection well stream from the `compi` field of the wells:

```
function c = getPhaseComponentFractionInjection(component, model, state, force)
    % Get the fraction of the component in each phase (when
    % injecting from outside the domain)
    c = cell(model.getNumberOfPhases(), 1);
    if isfield(force, 'compi')
        comp_{i} = vertcat(force.compi); % Wells
    else
        comp_{i} = vertcat(force.sat); % BC
    end
    index = component.phaseIndex;
    ci = comp_{i}(:, index);
    if any(ci ~= 0), c{index} = ci; end
end
```

Black-Oil Components

One perspective on the black-oil model is that it is a stepping stone from the immiscible model toward more compositional behavior. The number of phases still equals the number of components and each component is paired with a corresponding phase of the same name. The water component is considered immiscible, and we do not need to modify our implementation:

$$\boldsymbol{\rho}_{w,\alpha} = \begin{cases} \boldsymbol{\rho}_{\alpha}, & \text{if } \alpha = w, \\ 0, & \text{otherwise.} \end{cases} \tag{5.12}$$

There is a symmetry to the two components when $\mathbf{R}_s$ and $\mathbf{R}_v$ are both present:

$$\boldsymbol{\rho}_{o,\alpha} = \begin{cases} \boldsymbol{\rho}_o^S \mathbf{b}_o, & \text{if } \alpha = o, \\ \boldsymbol{\rho}_o^S \mathbf{b}_g \mathbf{R}_v, & \text{if } \alpha = g, \\ 0, & \text{otherwise,} \end{cases} \qquad \boldsymbol{\rho}_{g,\alpha} = \begin{cases} \boldsymbol{\rho}_g^S \mathbf{b}_o \mathbf{R}_s, & \text{if } \alpha = o, \\ \boldsymbol{\rho}_g^S \mathbf{b}_g, & \text{if } \alpha = g, \\ 0, & \text{otherwise.} \end{cases} \tag{5.13}$$

Mathematically, the black-oil model is an extension of the immiscible flow model, and the oil and gas components thus inherit from the immiscible implementation. We only examine the oil component; the gas component is essentially analogous:

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

```
function c = OilComponent(name, gasIndex, disgas, vapoil)
    c@ImmiscibleComponent(name, gasIndex);
    [c.disgas, c.vapoil] = deal(disgas, vapoil);
    c = c.functionDependsOn('getComponentDensity', ...
                            'ShrinkageFactors', 'PVTPropertyFunctions');
    if vapoil
        c = c.functionDependsOn('getComponentDensity', 'rv', 'state');
    end
end
```

Next, we define the component density for the oil component in vaporized oil as the product of $\mathbf{R}_v$ together with the oil shrinkage factor and the oil surface density. We must also take care to modify the density in the oil phase to account for the swelling due to dissolved gas per (5.13). The actual value of $\mathbf{R}_s$ is not explicitly used, but it implicitly impacts the results via the evaluation of the shrinkage factors themselves, as we saw in Figure 5.5:

```
function c = getComponentDensity(component, model, state)
    c = getComponentDensity@ImmiscibleComponent(component, model, state);
    if component.disgas || component.vapoil % Check for black-oil behavior
        b = model.getProps(state, 'ShrinkageFactors');
        phasenames = model.getPhaseNames();
        oix = (phasenames == 'O');
        reg = model.PVTPropertyFunctions.getRegionPVT(model);
        rhoOS = model.getSurfaceDensities(reg, oix);
        if component.disgas % Component density is not phase density
            bO = b{oix};  c{oix} = rhoOS.*bO;
        end
        if component.vapoil % There is mass of oil in gaseous phase
            gix = phasenames == 'G';
            rv = model.getProp(state, 'rv');
            c{gix} = rv.*rhoOS.*b{gix};
        end
    end
end
```

Two-Phase Compositional Components

For the `compositional` module discussed in more detail in Chapter 8, we assume we have up to two phases present that are formed through vapor–liquid equilibrium. Then the following holds:

$$\boldsymbol{\rho}_{i,\alpha} = \begin{cases} \mathbf{X}_{i,\ell}\boldsymbol{\rho}_{\ell}, & \text{if } \alpha = \ell, \\ \mathbf{X}_{i,v}\boldsymbol{\rho}_{v}, & \text{if } \alpha = v, \\ 0, & \text{otherwise.} \end{cases} \tag{5.14}$$

The definition of the component density is then straightforward from the mass fractions and mass density obtained from flashing the compositions:

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

```
function c = getComponentDensity(component, model, state, varargin)
    c = component.getPhaseComposition(model, state, varargin{:});
    rho = model.getProps(state, 'Density');
    for ph = 1:numel(c)
        if ~isempty(c{ph})
            c{ph} = rho{ph}.*c{ph};
        end
    end
end
```

5.4.3 Temporal Discretizations

Combining automatic differentiation with the state-function framework enables you to decouple functions from their derivatives. One advantage of this approach is that it becomes easy to implement different temporal discretizations. The default discretization in MRST is always implicit, but it is not difficult to change which variables and functions are evaluated implicitly (i.e., based on the yet unknown state at the end of the timestep).

Let us consider the discrete component flux from (5.3), where the flux is made up of a potential term $\mathbf{\Theta}_\alpha$, the absolute transmissibility $\mathbf{T}_f$, and the component mass mobility $\boldsymbol{\lambda}_{i,\alpha}$. We assume that the transmissibility is a static quantity. We also observe that the rapid speed of propagation for pressure means that the potential difference should be evaluated implicitly:

$$\mathbf{V}_{i,\alpha} = -\boldsymbol{\lambda}_{i,\alpha}^{f,k}\mathbf{T}_f\mathbf{\Theta}_\alpha^{n+1}. \tag{5.15}$$

The component mobility is here evaluated at time level k, so that $k = n$ gives explicit discretization of mobility and $k = n + 1$ results in a fully implicit scheme. We refer the reader to subsection 9.3.4 of the MRST textbook [3] for a discussion on explicit versus implicit schemes in general and to Section 10.2 for examples of sequential-implicit and sequential-explicit solvers for incompressible flow.

If we have the states at both the current and the previous time level as `state` and `state0`, respectively, the explicit flux ($k = n$) on the form (5.15) can be implemented as two state-function evaluations, here given for a single phase:

```
mob  = model.getProp(state0, 'FaceMobility');                % Previous time
kpot = model.getProp(state, 'PermeabilityPotentialGradient'); % Current time
v    = -mob{1}.*kpot{1};                                     % Explicit/implicit
```

Implementing standard explicit and implicit schemes in this way is straightforward. However, the two state functions are made up of many individual state functions for viscosity, relative permeability, density, and so on. If we want to exert fine-grained control over the implicitness of each individual function or have a function with implicitness that varies spatially, we need a more systematic approach. For this

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

reason, discrete fluxes are implemented in the `FlowDiscretization` class. This class contains an instance of a `FlowStateBuilder` class, which serves a dual purpose:

1. Before a timestep is simulated by the simulator, the flow-state builder may modify the suggested timestep to ensure stability of the underlying discretization scheme. You may already know the timestep selector classes from the MRST textbook [3, subsection 12.3.3], which select timesteps based on a variety of heuristics to target a certain accuracy or number of nonlinear iterations. The timestep limit imposed by the flow-state builder should ensure stability of the resulting scheme. For an explicit treatment of the mobility, the Courant–Friedrichs–Lewy (CFL) condition limits stability.
2. During a linearization, the flow-state builder will build a hybridized `state` that contains a mixture of implicit (AD, current time level) and explicit (double, from previous time level) variables. The hybridized state is used to evaluate the state functions that are required to compute the fluxes:

   ```
   v = model.getProp(flowstate, 'PhaseFlux')
   ```

   The hybridized state contains a combination of the values at the current timestep and the values at the previous timestep for all state variables, making the scheme either explicit, implicit, or somewhere in between. The calling signature acts directly on the state, automatically carrying forward any AD variables:

   ```
   fd = model.FlowDiscretization;
   flowstate = fd.buildFlowState(model, state, state0, dt); % Hybridize state
   ```

The functionality just discussed is implemented through two member functions of the `FlowDiscretization` class: `buildFlowState` creates the hybridized state struct and `getMaximumTimestep` calls the corresponding functions from the flow-state builder itself.

Fully Implicit

The default implementation of the fully implicit flow-state builder is trivial. The theoretical maximum timestep for an unconditionally convergent scheme is infinity, whereas the flow state itself is just the state at the current timestep:

```
function dt = getMaximumTimestep(fsb, fd, model, state, state0, dt, forces)
    dt = inf; % No time-step restriction
end
function flowState = build(builder, fd, model, state, state0, dt)
    flowState = state; % Fully implicit
end
```

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

The utility function `setTimeDiscretization` modifies the flux discretization of an existing model and you can use it to switch between the flow-state builders. Fully implicit is the default choice, but it is still possible to explicitly select it:

```
model_fim = setTimeDiscretization(model, 'fully-implicit'); % FIM
```

Explicit

Explicit schemes are attractive for their reduced numerical diffusion and relatively low computational cost for hyperbolic equations. They are, however, only conditionally stable in that the timestep is limited by the CFL condition. The general condition for a conservative discrete flux is found in equation (9.14) of the MRST textbook [3], but we can rewrite it slightly to use the maximum derivative of the fractional flow f together with a fixed total velocity v_t:

$$\eta = \frac{\Delta t}{\Delta x} \max_S |f'(S)| v_t \leq 1. \tag{5.16}$$

Explicit schemes update the values of a cell based on the values of its neighbors at the previous time level. This means that the timestep should be limited so that the fastest wave entering a cell should not exit that cell during the timestep or the stability will be impacted. The maximum derivative of the flux function must be determined over the interval spanned by the range of saturations inside the domain of dependence for each cell to obtain a *sharp* upper bound for the Courant number η and produce a scheme that is formally stable. (In practice, one often uses a somewhat stricter condition by maximizing over all saturation values found in the initial and boundary data, because this set can be determined once and for all.) Systems of equations do not generally fulfill a maximum principle, and sharp and rigorous upper bounds on η can become expensive to compute. Therefore, we instead compute numerical estimates and target a reduced upper bound on η to have some margin of safety. The numerical estimate is based on the form given in [9]. (Notice also that the upper stable limit on η generally depends on the discretization.)

If there are no capillary forces, we can write an estimate of the phase Courant number η_s for cell c in a three-phase system as

$$\eta_s[c] = \frac{\Delta t}{\mathbf{\Phi}[c]} \mathbf{\Lambda}[c] \sum_\gamma \mathbf{V}_t[\gamma]. \tag{5.17}$$

Here, the sum is taken over a set of faces (e.g., all faces γ that result in flow out from the cell) and $\mathbf{\Lambda}[c]$ is the largest eigenvalue of the Jacobian

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

$$\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{S}} = \begin{bmatrix} \frac{\partial f_w}{\partial S_w} & \frac{\partial f_w}{\partial S_o} \\ \frac{\partial f_o}{\partial S_w} & \frac{\partial f_o}{\partial S_o} \end{bmatrix}$$

for the fractional-flow function in cell c, where the fractional flow is defined in the usual manner $f_\alpha = \lambda_\alpha / \sum_\beta \lambda_\beta$. If we consider a scalar problem in 1D with unit porosity, $\mathbf{\Lambda}$ becomes the largest derivative, the pore volume becomes $1/\Delta x$, and we recover (5.16). We also define the estimated component Courant number η_z from an approximate volumetric flux for each component, given as the ratio of the component mass flow and the cell component mass:

$$\eta_z[c] = \frac{\Delta t}{\mathbf{\Phi}[c]} \max_{i \in \{1,\dots,N\}} \left[\frac{1}{\mathbf{M}_i[c]} \sum_\gamma \mathbf{V}_i[\gamma] \right]. \tag{5.18}$$

The smallest of the two limits is chosen to set the maximum timestep; i.e., $\eta = \min(\eta_s, \eta_z)$. Both of these values are only estimates, because the eigenvalues and component masses are calculated at a specific set of saturations and compositions. In addition, the fluxes may change during the timestep. For these reasons, it is natural to target a lower Courant number than unity in practice when choosing timesteps. The target can be adjusted on a per case basis; different authors [2, 12] have noted that a target of 1.5 or 2 can be safe under certain conditions.

If we now return to the implementation of the explicit flow-state builder, we see that there are a number of options to set the reduced CFL target:

```
saturationCFL  = 0.9;  % Target saturation CFL. Should be <= 1 for stability.
compositionCFL = 0.9;  % Target composition CFL. Should be <= 1 for stability.
explicitFlux = {'FaceMobility', 'FaceComponentMobility',...
                'GravityPotentialDifference'}; % Explicit in FlowDiscretization
explicitFlow = {};     % Explicit quantities in FlowPropertyFunctions
explicitPVT = {};      % Explicit quantities in PVTPropertyFunctions
explicitProps = {};    % setProp/getProp exposed properties that should be explicit
initialStep = 1*day;   % Time step used if no fluxes are present
useInflowForEstimate = false;
```

In addition, the class allows us to specify explicit terms, which are either named functions from the different state-function groups in the reservoir model or properties in state, specified in `explicitProps`. These are most easily understood if we reformulate (5.1) as a difference of masses by removing source terms and weighting by the ratio of pore volume to timesteps and assume that the flux $\mathbf{V}$ is given by a general function F,

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

$$\mathbf{M}^{n+1} - \mathbf{M}^n + \frac{\Delta t}{\mathbf{\Phi}}\mathtt{div}\Big(F(\mathbf{u}_1^n, \mathbf{u}_2^{n+1}, \mathbf{G}_1^n, \mathbf{G}_2^{n+1})\Big) = 0. \tag{5.19}$$

The function F is parameterized by a set of state variables $\mathbf{u}_1$ and $\mathbf{u}_2$ that are evaluated explicitly and implicitly, respectively. Here, $\mathbf{u}_1$ would be the `explicitProps` and $\mathbf{u}_2$ contains the remainder of the state variables. We also assume that state functions $\mathbf{G}_1, \mathbf{G}_2$ are direct inputs in the same manner. Here, $\mathbf{G}_1$ is set by `explicitFlow` for flow properties and `explicitPVT` for PVT properties, `initialStep` specifies the maximum timestep at the start of the simulation when flux estimates are not present, and `useInflowForEstimate` provides a toggle for changing the definition of the faces in the sum in (5.17) from outflow to inflow faces.

The implementation of the hybrid state itself consists of two major parts: First, one replaces implicit state properties with the named explicit properties and then inserts the values of explicit property functions into the state cache so they are not recomputed. We use the implicit state as a starting point and use `setProp` to overwrite any specified variables:

```
flowState = state; % Everything is implicit by default
for i = 1:numel(builder.explicitProps)
    p = builder.explicitProps{i};
    flowState = model.setProp(flowState, p, model.getProp(state0, p));
end
for i = 1:numel(props)
    prop = props{i}; % Props for current container
    if ~isempty(state0.(name).(prop)) % Name is the current container
        state0.(name).(prop) = []; % Remove cached entries
    end
    f0 = model.getProps(state0, prop);
    flowState.(name).(prop) = f0;
end
```

At this point, we have a state in which all values are evaluated either explicitly or implicitly. Evaluating the state functions that depend on several values may thus result in partially explicit evaluations; e.g., the default flux will evaluate the gradient of pressure implicitly and treat the mobility explicitly. You can select the explicit flow-state builder with a single line for any generic model class:

```
model_exp = setTimeDiscretization(model, 'explicit'); % IMPES
```

Adaptive Implicit

The explicit strategy has less numerical diffusion than the standard implicit method and can make the discrete equations nearly linear but will usually result in severe

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

timestep restrictions in practice. A somewhat less restrictive approach is to take the timestep as given and instead select the implicitness on a cell-by-cell basis according to the CFL limit. We can define a rule that defines the mobility in each cell from the estimate:

$$\lambda_{i,\alpha}^{k}[c] = \begin{cases} \lambda_{i,\alpha}^{n+1}[c], & \text{if } \eta[c] > \eta_{\max}, \\ \lambda_{i,\alpha}^{n}[c], & \text{otherwise.} \end{cases} \tag{5.20}$$

This is the basic principle of adaptive-implicit methods (AIM) [1, 11], in which each property can be implicit or explicit in different parts of the domain. The adaptive-implicit flow-state builder inherits directly from the explicit version. The major difference is that it produces no limits on the timestep but instead uses the estimated CFL number to set an `implicit` flag for each cell in the domain. Building the flow state is then similar to the explicit version, with the assignment from the previous state for explicit properties replaced with a combination of explicit and implicit values:

```
f_hyb = implicit.*f + ~implicit.*f0;
flowState.(name).(prop) = f_hyb;
```

We remark that this implementation is somewhat inefficient, because the property may be computed twice. Because we have a cell-wise indicator for the implicit flag, AIM currently relies on introducing implicitness via state functions in the PVT and flow groups that give cell-wise output only, because the flag is set on a per cell basis. Values on the face in the flux discretization are assumed to be computed from the cell-wise values. Note that the coupling between wells and perforated reservoir cells is evaluated implicitly regardless of CFL condition, because these cells are assumed to be subject to rapid changes in flow conditions. AIM is selected by specifying the values `'adaptive-implicit'` or `'aim'`:

```
model_aim = setTimeDiscretization(model, 'adaptive-implicit'); % AIM
```

5.4.4 Example: Fully Implicit, Explicit, and Adaptive Implicit

The choice of temporal discretization determines both the computational efficiency and how accurate a numerical scheme will resolve sharp displacement fronts. To illustrate this, we consider a simple advection problem, $\phi S_t + a S_x = 0$, corresponding to two-phase flow with equal phase viscosities and linear relative permeabilities. The CFL condition for this equation is $(a\Delta t/\phi\Delta x) \leq 1$. Subsection 9.3.4 in the MRST textbook [3] presents a discussion of truncation errors and numerical smearing of the explicit and implicit schemes for the case of $\phi = 1$. The discussion

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

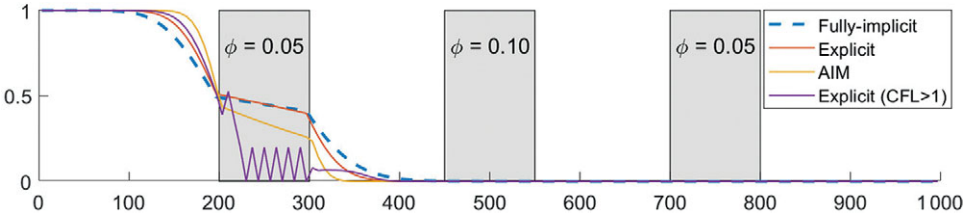


Figure 5.8 Four different solvers: fully implicit, explicit stable, explicit unstable, and adaptive-implicit saturation fronts, together with the varying porosity field for an advection equation. The implicit timesteps have CFL less than unity in the white region where $\phi = 0.5$ but exceed 1 in the grey regions.

will be made easier by considering a one-dimensional problem but, as we will see later, the approach is the same for more complex problems. This example is found in MRST as `immiscibileTimeIntegrationExample`.

We set up the advection equation by choosing equal phase viscosities and linear relative permeability curves for a two-phase immiscible fluid model. We let the porosity vary one order of magnitude throughout a domain of length $L = 1\,000$ m

$$\phi(x) = \begin{cases} 0.05, & \text{if } x/L \in [0.20, 0.30] \cup [0.70, 0.80], \\ 0.10, & \text{if } x/L \in [0.45, 0.55], \\ 0.50, & \text{otherwise.} \end{cases} \tag{5.21}$$

The average fluid velocity a per volume is constant throughout the domain so that a reduced porosity leads to much higher velocity in the pores, in turn severely limiting the length of stable timesteps for an explicit solver. The varying porosity is plotted in Figure 5.8 together with the solutions.

Default timesteps for fully implicit and adaptive-implicit methods are chosen to give a Courant number of 0.75 in the high-porosity cells. The explicit scheme selects timesteps automatically based on an estimate of the actual Courant number, which is 5 and 10 times higher in the low-porosity cells and will therefore use more timesteps locally as the front passes through these cells. We can adjust the CFL target used to select timesteps for the explicit flow-state builder up from the default of 0.9, at the risk of introducing unphysical oscillations in the solution.

To demonstrate instabilities that occur when exceeding the CFL limit, we increase the limit to five and disable convergence checks by setting a flag to indicate that the model is linear (i.e., that it is sufficient to only iterate once in the Newton solver):

```
model_explicit_largedt = setTimeDiscretization(model, 'explicit', ...
                'verbose', true, 'saturationCFL', 5); % Potentially unstable!
model_explicit_largedt.stepFunctionIsLinear = true;   % No convergence check
```

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

It is also possible to adjust whether specific properties will be evaluated explicitly or implicitly through the properties of the flow-state builder itself:

```
fsb = model_explicit.FlowDiscretization.getFlowStateBuilder();
disp(fsb)
```

```
explicitFlux:  {'FaceMobility'  'FaceComponentMobility'
                'GravityPotentialDifference'}
explicitFlow:  {}
explicitPVT:   {}
explicitProps: {}
```

Once we have chosen temporal discretization for each property, we can simulate as usual. We can pass a `'verbose'` argument larger than one when setting the time discretization to get additional output during the simulation from the flow-state builder. For example, looking at the output from the explicit solver, we get a message at the beginning of each timestep:

```
Solving timestep 018/158: 30 Days -> 33 Days, 8 Hours
Time-step limited by saturation CFL: 3.33 Days reduced to
       9.66 Hours (87.92% reduction)
```

The global timestep will be limited by the cell experiencing the largest Courant number. Here, this means that a single timestep of 3.33 days was split into several substeps that end at the next report step.

We next simulate the same case with the AIM solver. When we monitor the progress, we now observe that the message has changed from a notification of reduction in timestep to a notice of how many cells are implicit:

```
Solving timestep 050/158: 136 Days, 16 Hours -> 140 Days
Adaptive implicit: 44 of 150 cells are implicit (29.33%).
0 limited by composition, 44 limited by saturation, 0 belong to wells.
```

Figure 5.8 shows the solutions. The fully implicit method (FIM) and AIM both use the prescribed 158 timesteps, whereas the explicit solver uses 1 357 and 307 steps with stable and unstable Courant number, respectively. Of these two, only the stable setup avoids oscillations near the low porosity regions. FIM is the least accurate, which is typical for linear waves, because these are highly susceptible to numerical diffusion. For more details of the impact of numerical diffusion in practical simulation, see Chapter 7. You may note that the explicit solver introduces somewhat more smearing than AIM. Some may find this to be counter intuitive, but it follows directly from the truncation error analysis presented in the MRST textbook [3, subsection 9.3.4]: Numerical smearing decreases with decreasing timesteps for implicit methods but increases for explicit methods. A similar effect is seen in Subsection 3.4.2.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

We finally note that this approach to the explicit solver is somewhat inefficient, because the residual is computed and the full Jacobian is assembled in the entire domain. An alternative approach is to use the `PressureModel` and `TransportModel` from the `sequential` module to solve a separate pressure equation and adjust the temporal discretization in the transport solver. Although we omit it here for brevity, this is demonstrated in the example. A companion example, `blackoilTimeIntegrationExample`, demonstrates how to simulate with different temporal discretizations for the SPE 1 and SPE 9 benchmark models. We encourage you to experiment with the different options to see the impact on accuracy and efficiency.

5.4.5 Spatial Discretizations

It is also possible to change the spatial discretizations used by the simulator. The terms of the governing equations are themselves broken down into a number of different state functions that can be replaced in the same manner as we did when switching the pore volume definition from static to dynamic in Subsection 5.3.3. A detailed description of different discretization techniques is outside the scope of this chapter, but we will consider two examples that demonstrate how to use alternate schemes in the framework: a multipoint flux-approximation scheme that is consistent for grids lacking K-orthogonality and a weighted essentially nonoscillatory (WENO) scheme for more accurate resolution of the component transport.

Multipoint Flux Approximation Scheme

By default, the AD simulators in MRST discretize the Darcy flux using the two-point flux approximation (TPFA), which is not consistent for general grids and permeability fields. Chapter 6 of the MRST textbook [3] describes these consistency issues and why the inconsistent scheme is still used for most practical reservoir simulation. Our focus here is how to integrate an already implemented scheme in simulators that use state functions.

As an example, we use the MPFA-O multipoint flux approximation scheme from the `mpfa` module, as described in [3, section 6.4] under the name "local-flux mimetic method." This can be integrated into the AD-OO framework by altering the definition of the phase-potential difference $\boldsymbol{\Theta}_\alpha$. The reason for the modification of this function in particular is twofold:

1. We would like to simulate multiphase flow, for which the phase-potential difference in (5.4) is used to upwind transported quantities in (5.7). Modifying the

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

potential ensures that all functions that make use of the direction of the discrete potential difference consistently see the correct direction.

2. Face-based transmissibilities are deeply entrenched in reservoir simulation and may be modified in many ways that may not fit neatly in the mathematical description of the problem. Examples include fault multipliers and pressure-dependent transmissibilities. A rigorous mathematical treatment of such effects must be done on a case-by-case basis (see, e.g., the fault treatment in [6]). We would like to incorporate these effects even when not using a TPFA scheme by retaining the concept of transmissibilites for each face. In this way, multipliers or pressure-dependent transmissibility can seamlessly be combined with the MPFA scheme and give reasonable results.

Note that our approach to change the spatial discretization uses the same machinery for state functions as we used to implement pore volumes and densities earlier. The modified form of (5.4) is written in terms of a matrix M_Δ that represents the discrete gradient required for the MPFA scheme and a matrix M_g for the treatment of hydrostatic head difference:

$$\boldsymbol{\Theta}_\alpha = M_\Delta(\mathbf{p}_\alpha) - gM_g\texttt{favg}(\boldsymbol{\rho}_\alpha)\texttt{grad}(\mathbf{z}). \tag{5.22}$$

The matrix M_Δ is identical in action to the discrete gradient operator `grad` for K-orthogonal grids but introduces additional nonzero entries in the pressure Jacobian that connect cells to their nodal neighbors. The values and sparsity pattern of the matrix take on different values for skewed grids or anisotropic permeability fields. By the same logic, M_g is an identity matrix for K-orthogonal grids.

Once we have the desired potential difference, the fluxes in (5.3) can be computed in a straightforward manner. Equation (5.22) is introduced to the simulator via a helper utility, `setMPFADiscretization`. This function is quite compact and relies mostly on the multipoint transmissibility calculator from the `mpfa` module for the heavy lifting, leaving only a few scaling operations to match the conventions in AD-OO for signs and gradients. For brevity, we only offer the necessary lines without detailed explanation:

```
require mpfa
[~, M] = computeMultiPointTrans(model.G, model.rock); % From MPFA code
Tv = M.rTrans;                       % Transmissibilities: cells -> inner faces
Tg = M.rgTrans(model.operators.internalConn, :);  % Gravity contributions

% Change sign and rescale operators to fit with AD-OO definition of gradient.
scale    = -1./(2*model.operators.T);
MPFAGrad = Tv.*scale;
Mg       = -Tg.*scale/2;
```

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

Integrating existing functionality into state functions can be an efficient way to extend a simple prototype implementation to more complex flow scenarios. After the MPFA transmissibilities are computed, we set up the state functions:

```
model = model.setupStateFunctionGroupings();
```

The scheme is then implemented by retrieving two state functions from the model and modifying them: First, the pressure gradient is given a custom gradient operator consistent with the MPFA scheme:

```
% Discrete gradient
fd = model.FlowDiscretization;
dp = fd.getStateFunction('PressureGradient');
dp.Grad = @(x) MPFAGrad*x;
fd = fd.setStateFunction('PressureGradient', dp);
```

Then, the gravity potential difference is modified with M_g as a weighting matrix:

```
% Gravity potential difference
dg = fd.getStateFunction('GravityPotentialDifference'); dg.weight = Mg;
model.FlowDiscretization = fd.setStateFunction('GravityPotentialDifference', dg);
```

Example: MPFA vs TPFA

The script `MPFAvsTPFAwithADExample` from `ad-core` demonstrates the use of MPFA on example 6.1.2 from the MRST textbook [3]. The example describes a homogeneous reservoir with a symmetric well pattern consisting of a single injector and two producers. To demonstrate typical grid-orientation effects on skewed grids, the grid is intentionally angled toward one of the two producers, as seen in Figure 5.9. Once we have set up a standard two-phase flow model and loaded the `mpfa` module, we can replace the TPFA discretization:

```
mrstModule add mpfa                          % Load module
model_mpfa = setMPFADiscretization(model); % Replace discretization
```

We first apply TPFA with both explicit and implicit time integration. We know that the water cut between the two producers should be symmetric for a consistent solver, but we observe a significant discrepancy in Figure 5.10. Using a consistent scheme results in a greatly improved balance between the wells as seen in the right plot of the same figure. Switching to an explicit scheme sharpens the production curves somewhat in both cases. There are multiple error sources at work,

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

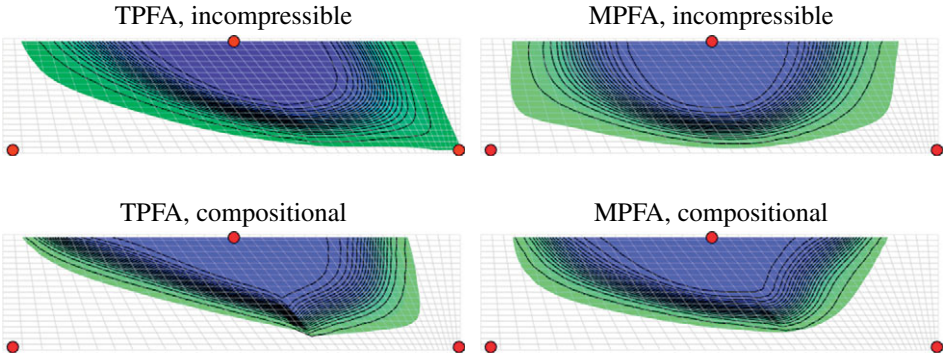


Figure 5.9 Two variants of the same symmetric scenario realized with an inconsistent method (left column) and a the consistent method (right column). The upper row is an incompressible and immiscible scenario, whereas the lower row shows a compositional scenario in which CO_2 displaces decane. Colors give isocontours for the implicit solution, and black lines represent the same isocontours for the explicit scheme.

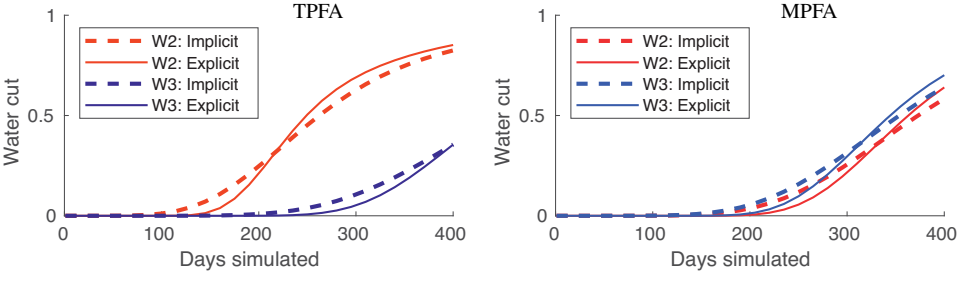


Figure 5.10 Water cut for the symmetry test for two different spatial discretizations: the inconsistent TPFA method (left) and the consistent MPFA method (right). The two producers should have equal water cut, but the skewed grid affects the results. Switching to a more accurate temporal discretization reduces the numerical diffusion but does not impact the asymmetry.

and selecting the solution with the least error requires a careful examination. In Figure 5.9 it is clear that the explicit discretization reduces the numerical diffusion near the water front, whereas the consistent discretization governs *where* the water front itself moves by producing a better velocity field.

A second variation of this scenario (set `useComp=true`) is a compositional model setup in which we inject supercritical CO_2 to displace a resident heavy component. With a mobility ratio different from unity, the values of the pressure matter directly for the results and not just indirectly through the velocity field.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

The higher mobility of the injected gas results in stronger grid-orientation effects, as is clearly exhibited in the lower row of Figure 5.9. Note that we did not have to change any of the MPFA-related routines to simulate a very different type of flow physics.

High-Resolution Scheme for Upwinding

Fully implicit methods are popular in reservoir simulation due to their unconditional stability. Unfortunately, when paired with a single-point upwind (SPU) scheme on the form (5.7), the accuracy of the scheme can be poor for flux functions that are not self-sharpening. The previous section showed that varying the degree of implicitness for saturations and compositions can reduce the numerical diffusion of the SPU scheme. A complementary option to varying implicitness is changing the discretization itself; e.g., by replacing SPU by the high-resolution discontinuous Galerkin methods described in Chapter 3. We consider an alternative WENO discretization [5] in which states used to compute the flux across each cell interface are reconstructed point-wise from a number of polynomial interpolations from the cell-averaged values in the neighboring cells. In short, this scheme computes a second-order approximation from several truncated Taylor expansions around the upstream cell for each interface f,

$$\boldsymbol{\lambda}_\alpha^f = \boldsymbol{\lambda}_\alpha[\ell] + \sum_j w_{j,\ell} \boldsymbol{\sigma}_{j,\alpha}^\ell \left(\mathbf{x}_f - \mathbf{x}_{C_1(f)}^\ell\right), \quad \begin{cases} \ell = C_1(f), & \text{if } \boldsymbol{\Theta}_\alpha[f] \le 0, \\ \ell = C_2(f), & \text{otherwise.} \end{cases} \tag{5.23}$$

Here, each approximate gradient $\boldsymbol{\sigma}_{j,\alpha}^\ell$ is determined from the values of the phase mobility at the centers of a triplet of cells (in the 2D case) that includes ℓ and the sum runs over all such triplets that contain the upstream cell. A nonzero weight $w_{j,\ell}$ is assigned to each interpolated value, with $\sum_j w_{j,\ell} = 1$. These are nonlinear functions that depend on the cell-averaged values, and small weights are assigned to cell triplets (i, j, ℓ) giving large gradients to avoid spurious oscillations near discontinuities in the solution (see Lie et al. [4] for more details). The scheme is second-order accurate away from discontinuities and applicable to fairly general grids. The original paper only discussed the fully implicit version of the scheme but, as we have seen, switching to explicit or AIM is just a matter of changing the flow-state builder.

Example: Combining WENO and AIM

The example `wenoExampleAD.m` demonstrates the WENO scheme on a simple quarter five-spot example in which a tracer fluid is injected. Due to the linear

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

flux function, the front is highly sensitive to numerical diffusion. Just as when we switched from TPFA to MPFA, the switch from SPU to WENO is done by a helper routine:

```
model_weno = setWENODiscretization(model);
```

The routine essentially consists of two parts. The first is setting up the WENO discretization as an `UpwindDiscretization` subclass (see the class documentation for more details on available options):

```
weno = WENOUpwindDiscretization(model);
```

In addition, the class has a member function `faceUpstream` that determines face values from the cell values. Several state-function classes that produce outputs on each face can contain an upwind discretization class. If none is passed, the default function from the model's `operators` struct is used instead. Once the WENO scheme has been set up, our helper routine can update the definitions of $\boldsymbol{\lambda}_{\alpha}$ and $\boldsymbol{\lambda}_{i,\alpha}$:

```
p = model.FlowDiscretization; % Get default
p = p.setStateFunction('FaceMobility', FaceMobility(model, weno));
p = p.setStateFunction('FaceComponentMobility', FaceComponentMobility(model, weno));
model.FlowDiscretization = p; % Replace
```

The advantage of the modular approach provided by state functions is clear: We can easily modify any kind of upwinded function with a WENO scheme, just as we could easily switch to AIM for any type of model. We can now simulate a quarter five-spot injection scenario on a 50×50 grid with four different schemes: FIM with SPU, FIM with WENO, AIM with SPU, and AIM with WENO. Figure 5.11 uses the same style of plot as the previous example and shows that the fully implicit SPU scheme is very diffusive. The white outline in the plot of Courant numbers highlights cells near the wells that are evaluated implicitly in the AIM scheme. Even when many of the cells are treated explicitly, there is significant smearing of the front. If we instead switch to WENO, the fully implicit solver is comparable in accuracy to the explicit SPU. Switching to AIM for the WENO solver results in significant additional improvement. Obtaining accurate results for scenarios that are sensitive to smearing of fronts may therefore require a combination of a fine grid, high-resolution schemes, tailored time integration strategies, and carefully chosen time steps, all of which are possible with the new AD-OO framework in MRST.

https://doi.org/10.1017/9781009019781.010 Published online by Cambridge University Press

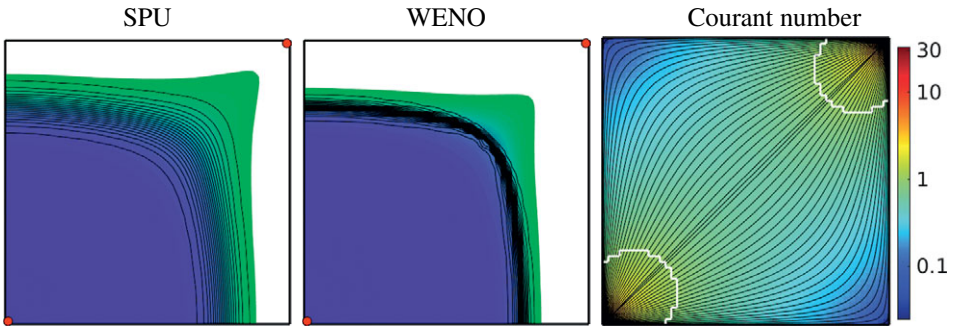


Figure 5.11 A quarter five-spot example with a linear displacement. The linear wave is sensitive to numerical diffusion and we observe significant smearing for the implicit SPU scheme (left) even when using AIM (black lines). Smearing diminishes significantly when switching to implicit WENO (middle) and is further reduced by applying AIM (black lines). Courant numbers (right) exceed the stability limit of explicit schemes in the near-well regions, where many streamlines depart the well. AIM therefore uses an implicit discretization in the regions inside the white lines.

5.5 Concluding Remarks

The state-function framework, together with the modular initialization of AD variables, represents a significant step forward in the capability of the AD-OO prototyping environment. Altogether, the combination of new features makes it easy to modify, extend, and understand simulators for highly complex processes in porous media. The design of the simulator as a graph of loosely coupled, replaceable functions is a step forward from the flexibility afforded by automatic differentiation, and many of the other chapters in this book build upon these features to support a range of physical effects, including chemical enhanced oil recovery (Chapter 7), compositional flow (Chapter 8), flow in fractured reservoirs (Chapters 10 and 11), and geothermal flow (Chapter 12).

References

[1] H. Cao. Development of techniques for general purpose simulators. PhD thesis, Stanford University, Stanford, CA, 2002.
[2] K. H. Coats. A note on IMPES and some IMPES-based simulation models. *SPE Journal*, 5(3):245–251, 2000. doi: 10.2118/65092-PA.
[3] K.-A. Lie. *An Introduction to Reservoir Simulation Using MATLAB/GNU Octave: User Guide for the MATLAB Reservoir Simulation Toolbox (MRST)*. Cambridge University Press, Cambridge, UK, 2019. doi: 10.1017/9781108591416.
[4] K.-A. Lie, T. S. Mykkeltvedt, and O. Møyner. A fully implicit WENO scheme on stratigraphic and unstructured polyhedral grids. *Computational Geosciences*, 24: 405–423, 2020. doi: 10.1007/s10596-019-9829-x.

[5] X.-D. Liu, S. Osher, and T. Chan. Weighted essentially non-oscillatory schemes. *Journal of Computational Physics*, 115(1):200–212, 1994. doi: 10.1006/jcph.1994.1187.

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

[7] A. S. Odeh. Comparison of solutions to a three-dimensional black-oil reservoir simulation problem (includes associated paper 9741). *Journal of Petroleum Technology*, 33(1):13–25, 1981. doi: 10.2118/9723-PA.

[8] R. Rin, P. Tomin, T. Garipov, D. Voskov, and H. A. Tchelepi. General implicit coupling framework for multi-physics problems. Paper presented at *SPE Reservoir Simulation Conference, Montgomery, TX, 20–22 February*. Society of Petroleum Engineers, 2017. doi: 10.2118/182714-MS.

[9] T. Russell. Stability analysis and switching criteria for adaptive implicit methods based on the CFL condition. In *SPE Symposium on Reservoir Simulation*. Society of Petroleum Engineers, 1989. doi: 10.2118/18416-MS.

[10] T. Tantau. Graph drawing in TikZ. In *International Symposium on Graph Drawing*, pp. 517–528. Springer, Berlin, 2012. doi: 10.1007/978-3-642-36763-2_46.

[11] G. W. Thomas and D. H. Thurnau. Reservoir simulation using an adaptive implicit method. *Society of Petroleum Engineers Journal*, 23(5):759–768, 1983. doi: 10.2118/10120-PA.

[12] L. C. Young and T. F. Russell. Implementation of an adaptive implicit method. In *SPE Symposium on Reservoir Simulation, New Orleans, LA, 28 February–3 March*. Society of Petroleum Engineers, 1993. doi: 10.2118/25245-MS.