6

Faster Simulation with Optimized Automatic Differentiation and Compiled Linear Solvers

OLAV MØYNER

Abstract

Many different factors contribute to elapsed runtime of reservoir simulators. Once the cases become larger and more complex, the required wait time for results can become prohibitive. This chapter discusses three features recently introduced into the object-oriented, automatic differentiation (AD-OO) framework of the MATLAB Reservoir Simulation Toolbox (MRST) to make simulation of large cases more efficient. By analyzing the sparsity pattern of the Jacobians for some of the most common operations involved in computing residual flow equations, we have developed different implementations of automatic differentiation that offer better memory usage and require fewer floating-point operations. Using these so-called AD backends ensures (much) faster assembly of linearized systems. Likewise, these systems can be solved much faster by utilizing external packages for linear algebra; herein, primarily represented by the AMGCL header-only C++ library for solving large sparse linear systems with algebraic multigrid (AMG) methods. Last, but not least, the new "packed problem" format simplifies the management of multiple simulation cases and enables automatic restart of simulations and an ability for early inspection of results from large batch simulations. Altogether, these features are essential if you are working with bigger simulation models and want timely results that persist across MATLAB sessions.

6.1 Introduction

The premise of this chapter is that you wish to use the MATLAB Reservoir Simulation Toolbox (MRST) to perform simulations that involve many thousands of unknowns. Maybe you have developed your own simulation scripts, or maybe your research involves one of the many preexisting ones in MRST. You have verified that your setup gives correct results for a smaller case, and it is time to run a much

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

bigger version to get fully resolved results for your paper, or you have simply read somewhere that MRST supports simulations of industry-standard complexity and want to try the software for a complex (real asset) model you have. You are probably aware that MRST is primarily written to be a flexible prototyping platform and, like many others, you may question whether it scales to large models and how well its runtime compares with commercial simulators or academic research codes written in a compiled language.

As a prototyping platform, MRST cannot and should not try to compete with dedicated high-performance simulators written in Fortran, C, or C++ in terms of computational speed. If given the choice between efficiency when *writing* code or efficiency when *executing* it, a prototyping platform should always tend toward ease of implementation. There is a caveat to this, however, because the value of a prototype solver is limited if it cannot run relevant test cases within a reasonable amount of time. There are several factors that potentially could cause MRST codes to be slow:

1. **Interpreted language:** You may have good reasons to suspect that any operation executed in an interpreted language is slower than in standard compiled languages like Fortran, C, or C++. In MATLAB, this is addressed by, among others, the just-in-time (JIT) compiler that translates scripts into machine code and by vectorization calling highly efficient libraries written in C++/C/assembly behind the scene. In MRST, we have been careful to utilize vectorization, logical indexing, preallocation of memory, etc., as discussed throughout the MRST textbook [14], to improve computational performance. In many cases, MRST scripts can therefore run almost as fast as if they were written in a compiled language. We believe that our experience in this regard is applicable in general.
2. **Computational overhead:** It is generally difficult to avoid introducing computational overhead when prototyping a new computational idea. In MRST, a primary contributor to this is the automatic differentiation that comprises a crucial part of the object-oriented, automatic differentiation (AD-OO) framework. Because AD is used extensively in the linearization and assembly of linear systems, which usually constitute a significant part of the total runtime of a simulator, it is important that the AD implementation is as computationally efficient as possible. In Section 6.2 we therefore introduce you to a family of new AD backends that have been optimized to utilize certain sparsity patterns in the computation and accumulation of derivatives. These backends can also potentially be MEX-accelerated. Although the discussion focuses on MRST, we highlight many issues and possible remedies that may be relevant to other vectorized languages.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

3. **Redundant computations:** When prototyping a new idea, avoiding redundant function and property evaluations may not be the first thing on your mind. These can easily become quite entrenched in your code and difficult to root out after the simulator has been validated. The modular state-function framework introduced in Chapter 5, with its associated cache mechanism and dependency graphs, has been designed to mitigate this problem. We believe that these ideas represent a very promising way forward to develop more versatile simulators, not only in MRST but also more generally.
4. **Linear algebra:** Hearsay has that the major fraction of the computational cost of a carefully designed reservoir simulator should come from the linear solver. MATLAB's direct linear solvers are highly efficient and hard to beat on small linear systems but do not scale well to larger reservoir models. Subsection 12.3.4 of the MRST textbook [14] therefore outlined how you can configure MRST to use more efficient (preconditioned) iterative solvers. In Section 6.3, we give an updated and more in-depth discussion of how you can use external iterative solvers, written in a compiled language, to ensure that the computational performance of MRST scales well also for (surprisingly) large models. The interfaces presented are specific to MRST, but the specific solution strategies and solvers discussed should be applicable to any reservoir simulator.

For completeness, we emphasize that there are also other sources of computational overhead that are not as simple to reduce. Per design, MATLAB has many convenience features that contribute to making the language an attractive prototyping platform. This includes dynamic types, dynamic allocation and deallocation, a lot of safety checks, and variable list of input–output parameters. There is also overhead associated with certain indexing operations in sparse matrices and calling compiled libraries. Likewise, the model for utilizing concurrency is not optimal and it may be difficult to avoid expensive cache misses. So, in sum: MRST can be made efficient also for (surprisingly) large models, but you should not generally expect that it can compete 100% with a compiled and highly optimized simulator.

Running a case with many cells and complex flow physics will nonetheless take time, regardless of the simulator, so you will often have to leave it running for several hours. Sometimes, the simulation is aborted because the timestep was cut too many times or you decide to abort it yourself because it takes too long to finish. You may also experience unwanted computer crashes or reboots during a long simulation. In either case, unless your setup has a restart mechanism, your results are gone, and you have no choice but to start the simulation again and hope that the same thing does not happen this time. If this sounds familiar, Section 6.4 was written for you; it describes recent functionality that enables automatic restarts for aborted or failed simulations.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

The discussion in this chapter does not contain any details about the underlying reservoir modeling, and to benefit from it you should therefore be familiar with how to set up and run simulations in MRST; this is described in the MRST textbook [14]. To a certain extent, features discussed later in this chapter can be automatically selected. If you are using utility functions such as `initEclipseProblemAD`, `selectLinearSolverAD`, or `getNonLinearSolver`, chances are that you are already using many of the features from this chapter.

Benchmark setup: All computational performance results reported later in this chapter, in terms of the runtime for individual operations and simulation steps, are from a Windows 10 workstation with a single Intel Xeon E3-1275 v6 and 64 GB of RAM. This CPU is a 2017 model that operates at 3.8 GHz with four cores. Some of the functionality discussed relies on MEX extensions that herein were compiled with the Visual Studio 2019 compiler. Appendix A gives you more details about how such MEX extensions are set up in MRST.

6.2 Accelerated Implementation of Automatic Differentiation

The literature about automatic differentiation is abundant, but if you are not familiar with the concept and how it can be used for numerical computations, we recommend the introductory article by Neidinger [18], which focuses on how this technique can be implemented efficiently in MATLAB. This paper was a strong source of inspiration when we first developed capabilities for automatic differentiation in MRST. In this section, we first explain the context for automatic differentiation in reservoir simulation, briefly outline the different ways it can be implemented, and then discuss what we have done to improve the efficiency of this technique in MRST.

The need for derivatives: The basic flow equations in reservoir simulation can be summarized as a set of equations that each model the discrete conservation for a quantity $\mathbf{M}$ with a corresponding flux $\mathbf{V}$ and source term $\mathbf{Q}$,

$$\frac{\mathbf{M}^{n+1} - \mathbf{M}^n}{\Delta t^n} + \mathtt{div}(\mathbf{V}) - \mathbf{Q} = 0. \tag{6.1}$$

This is a generalized form of the type of flow equations you will encounter in many of the chapters in Part III of the book; Chapters 7 and 8, for instance, present more details for the specific cases of chemical enhanced oil recovery (EOR) and compositional simulation. These resulting nonlinear equations are usually solved by a Newton–Raphson method, which requires the full set of derivatives with respect to

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

all primary variables to construct the necessary Jacobian matrix needed to compute an iterative increment.

In Chapter 5 we discussed the interdependencies of the many quantities that enter such a computation, explained how the computation can be modularized using state functions, and highlighted how the output from specific state functions often are used in several different equations simultaneously. For example, the total phase mobility in a given cell may enter into each of the component conservation equations, as well as in the equations for any wells connected to that cell. So, in reservoir simulation, unlike in many other applications that utilize automatic differentiation, the computation of derivatives is in a many-to-many context, in which we require the derivatives of the residual equations in all cells (and well completions/segments) with respect to all primary variables.

Forward versus reverse AD: One possible classification distinguishes between forward-mode and backward- or reverse-mode AD. In the former, the derivatives of each individual operation involved in evaluating a mathematical function are tracked simultaneously with the value of these operations using standard algebraic rules. The effect of the individual operations is accumulated using the well-known chain rule. The forward mode can be implemented by introducing types for dual numbers that hold values and derivatives or by operator overloading, depending on the language itself. In reverse-mode AD, the graph of operations leading to a final value is carefully recorded and values of any intermediate variables are stored in a forward sweep. This is followed by a backward phase that propagates back the derivatives again using the chain rule. This is what is commonly called *backpropagation* in the context of machine learning and is similar to the adjoint method [9] for sensitivities of a time-dependent reservoir simulation. In addition, there exist source-transformation approaches in which the source code of a program is examined by another program with the intent to generate additional source code that produces any required derivatives.

Which of the methods will be most computationally efficient depends on the characteristics of the problem. Forward AD needs to store intermediate values and derivatives and is therefore considered to be the most efficient when the number of independent variables (input) is much lower than the number of function values (output). Conversely, reverse AD only stores intermediate values and the dependency and will therefore be more efficient for cases with many variables and few function values, as is typically the case in machine learning.

Use of AD for reservoir simulation: As we have already seen, reservoir simulation is a many-to-many context. At first glance it may not necessarily be obvious

which of the two methods just outlined will be the most efficient, and different approaches have therefore been applied in the literature. The first implementation of AD we are aware of appeared in an early version of the commercial INTERSECT simulator [5]. Its use was not continued, and use of AD for reservoir simulation has instead primarily been pioneered by Stanford's AD-GPRS simulator [22, 23, 27], which builds on ADETL, a library for forward-mode AD implemented in terms of expression templates in C++ [25, 26]. Seeing the success of AD-GPRS, we decided to implement the same type of methods in MRST [2, 12, 17], using ideas from Neidinger [18] to develop a variable-first forward-AD library suited for vectorized operations in MATLAB. The same approach was subsequently implemented almost verbatim in the open-source simulator OPM Flow [21] but has later been superseded by a localized, cells-first forward-AD library that is more suited to the underlying C++ language. It is our belief that forward AD is the best choice for reservoir simulation, but we acknowledge that other authors argue for the use of reverse-mode AD [13].

Improving the efficiency of MRST: Much of the flexibility found in MRST would not be possible without automatic differentiation. For instance, all simulators that use state functions described in Chapter 5 rely on AD to compute the Jacobians of complex graphs that consist of discretizations, functional relationships, and thermodynamic quantities. The benefit of such a flexible framework is limited if the evaluation becomes too slow. The main advantage of the original AD implementation in MRST was its simplicity. However, in trying to make a vectorized library, we also made some choices that affect performance adversely. For this reason, MRST has recently been extended to include support for C/C++ accelerated and optimized AD implementations. In the rest of the section, we examine how you can utilize known sparsity patterns of certain operations to reduce their computational cost (and memory consumption). We then discuss how different types of specialized AD backends can be used in MRST to quickly assemble Jacobians for larger test cases and obtain performance that rivals that of compiled simulators on typical shared memory systems used for prototyping; e.g., on workstations and laptops. Parts of this discussion are specific to MRST and reservoir simulation, but there are also patterns and considerations that are generally applicable to any vectorized language and for applications other than reservoir simulation.

6.2.1 Different Backends for Automatic Differentiation

If you are familiar with the AD-OO framework, from the MRST textbook [14] or one of the many tutorial examples, you have already seen primary variables being

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

initialized with `initVariablesADI`, which is the initializer for the default AD implementation in MRST. In a recent reformulation of the AD-OO framework, we introduce the concept of an automatic differentiation *backend* to initialize primary variables, because this makes it easy to change the AD implementation by replacing the backend in use. AD-OO contains several variations of forward-mode automatic differentiation that, e.g., use different storage formats for accumulating derivatives.

Each backend is derived from the `AutoDiffBackend` base class that takes care of initialization and consistent conversion of doubles to AD variables. For example, assume that we have three vectors and have declared two of them to be independent primary variables with AD type:

```
x = rand(10, 1); y = rand(10, 1); z = rand(10, 1); % Three vectors
[x, y] = initVariablesADI(x, y);                   % Initialize x, y as AD
```

We can then convert the third to be the same type of AD variable as `x` and `y` – i.e., utilize the same AD implementation and have the same primary variables – and initialize it to have zero derivatives with respect to these primary variables:

```
z = double2ADI(z, x);          % z -> AD, dz/dx = 0, dz/dy = 0
```

This gives us a systematic way of converting (vectors of) doubles to AD variables. Converting doubles to AD is essential when working with vectorized code. Examples of usage are provided in Subsection 6.2.5. Alternatively, we can express the same operations by first instantiating a backend and then using standard interfaces:

```
backend = AutoDiffBackend();
[x, y]  = backend.initVariablesAD(x, y);
z       = backend.convertToAD(z, x);
```

These two listings are completely equivalent in results. The only difference is that in the second alternative we could replace `AutoDiffBackend` in the first line with another class to use another type of AD. All AD-OO simulation models in MRST automatically set up a backend upon construction, which is later used when linearizing equations. You can extract or set the associated backend as follows:

```
backend = model.AutoDiffBackend; % Get AD backend
model.AutoDiffBackend = backend; % Set AD backend
```

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

AD objects initialized from different backends are usually *not* directly compatible. For this reason, we recommend that you always use the backend from the model when possible.

In the following, we first demonstrate the difference between the standard *sparse AD* representation and the *diagonal AD* on a simple test problem, before outlining the available implementations and discussing their advantages and drawbacks.

6.2.2 Motivation for Different Types of AD Backends

It will be illustrative to work through a simple numerical example to understand why we have implemented several AD variants in MRST. To this end, we consider a vector version of the simple scalar equation used in the crash course on state functions in Chapter 5:

$$\mathbf{G}(\mathbf{x}, \mathbf{y}, \mathbf{a}, \mathbf{b}) = \mathbf{xy} + \mathbf{ab}. \tag{6.2}$$

To understand the specific details of the following discussion, you should read this section first.

The output of the function in (6.2) is itself a vector with length L. The element-wise nature of the operations[1] means that the calculation of entries i and j of $\mathbf{G}$ only depends on the corresponding entries i and j in $\mathbf{x}$, $\mathbf{y}$, $\mathbf{a}$, and $\mathbf{b}$. We have four vector inputs to $\mathbf{G}$, and to compute the partial derivatives with respect to all input parameters we let the vector $\mathbf{v}$ of primary variables correspond to these four vectors:

$$\mathbf{v} = [\mathbf{x}^T, \mathbf{y}^T, \mathbf{a}^T, \mathbf{b}^T]. \tag{6.3}$$

Setup of the problem: The example `showConceptualAD` computes $\mathbf{G}$ and a few related operations together with the corresponding derivatives in several different ways. We first describe the basic operations in terms of doubles, omitting the setup of the state functions themselves. We set up a state and assign to it a container for the group of state functions that represent $\mathbf{G}$ to enable caching:

```
[a,b,x,y] = deal(rand(n,1), rand(n,1), rand(n,1),  rand(n,1)); % n entries each
state0 = struct('a', a, 'b', b, 'x', x, 'y', y);               % Initial state
state = group.initStateFunctionContainer(state0);              % Set up storage
```

[1] Note that, as in Chapter 5, we abuse notation for vector multiplication so that the multiplication of two column vectors results in element-wise multiplication. In this way, we match the MATLAB `times` operator `x.*y`; i.e., $(\mathbf{xy})_i = \mathbf{x}_i \mathbf{y}_i$.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

We then define the four operations we will use to study computational performance of automatic differentiation:

Listing 6.1 *Example of four operations that are typical in simulators.*

```
G = group.get([], state, 'G');   % Compute G for all input values
G_sub  = G(1:100:n);             % Extract a subset: every 100th value
v      = 2*G_sub;                % Compute on the subset of values
G(1:100:n) = v;                  % Insert the modified subset values
```

In addition to calculating **G**, we extract every hundredth element, multiply the resulting subset with a scalar, and then reinsert the values into **G**. Operations on subsets of values are very common in simulators that couple different models together or when introducing source terms in parts of the domain. Note that because the index is *nonsequential*, the subset operation is more expensive in terms of memory access than with an index that would pick the same number of elements in sequence.

For each of these four operations, the script measures the average time taken over many repeated calls of the four operations. If we assume the cost of adding and multiplying two numbers to be the same, computing each entry of **G** consists of three floating-point operations (flops). For the AD part, the derivative of an addition is simply the addition of the derivatives of each argument. The derivative of a product of two numbers with known derivatives can be found by the product rule

$$\big(f(x)g(x)\big)' = f'(x)g(x) + f(x)g'(x). \tag{6.4}$$

Hence, the cost of computing the derivative of multiplication is three flops. For **G**, we must thus perform a total of seven flops for each derivative. Because we have four nonzero derivatives, computing all derivatives of **G** with respect to **v** requires at least 28 flops per entry in **G**, in addition to the three operations for the value itself. Using AD to compute values *and* derivatives hence requires roughly 10 times as many operations than for just doubles to only compute the values. Note that counting flops is just a part of the picture for modern computing, because memory locality, cache effects, and specialized processor features in practice will have large impacts on the execution speed. In the following comparison, however, we will for simplicity say that a new AD implementation is very promising if its runtime is less than 10 times that of computing the **G** values only (using plain doubles).

Local AD operations in reservoir simulation: Even though (6.2) is just a conceptual illustration, the local operations in Listing 6.1 are nonetheless relevant to

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Table 6.1 *Examples of values* $\mathbf{M}$ *can take in* (6.1) *depending on the type of model in use.*

Physics	$\mathbf{M}$	Interpretation	Module
Immiscible	$\boldsymbol{\Phi}\boldsymbol{\rho}_w\mathbf{S}_w$	Mass of water phase	`ad-blackoil`
Black oil	$\boldsymbol{\Phi}\boldsymbol{\rho}_g^s(\mathbf{b}_g\mathbf{S}_g + \mathbf{R}_s\mathbf{b}_o\mathbf{S}_o)$	Mass of gas pseudocomponent	`ad-blackoil`
Compositional	$\boldsymbol{\Phi}(\boldsymbol{\rho}_l\mathbf{X}_{i,l}\mathbf{S}_l + \boldsymbol{\rho}_v\mathbf{X}_{i,v}\mathbf{S}_v)$	Mass of component i	`compositional`
Thermal	$\boldsymbol{\Phi}\boldsymbol{\rho}_f\mathbf{u}_f + (1-\boldsymbol{\Phi})\boldsymbol{\rho}_r\mathbf{c}_r\mathbf{T}$	Thermal energy in rock & fluid	`geothermal`

reservoir simulation and are encountered often during the assembly of discretized residual equations of the form (6.1). For instance, $\mathbf{M}$ is typically an expression that closely resembles (6.2); Table 6.1 shows a few possible examples from MRST. Each specific $\mathbf{M}$ is generally composed of a series of functions such as phase density or heat capacity that are functions of the primary variables. These are then multiplied and added together to form the final term we are interested in. Many of the intermediate results will also be used to evaluate other terms in the flow equations. Efficient computation of Jacobians for local operations is therefore an important prerequisite for fast assembly. In the following, we describe how utilizing the diagonal structure of such Jacobians is a key to fast assembly in MRST.

Two different AD representations: We can perform the same operations by initializing AD variables to simultaneously compute the Jacobians with respect to $\mathbf{v}$:

```
[aAD, bAD, xAD, yAD] = initVariablesADI(a, b, x, y);
state0 = struct('a', aAD, 'b', bAD, 'x', xAD, 'y', yAD);
```

The rest of the operations proceed as before, with the final value of `G` being an `ADI` class object that contains a vector of values for `G` and a list of square and sparse Jacobian submatrices, one for each of the four input vectors:

```
disp(G)
```

```
  ADI with properties:
    val: [1000000x1 double]
    jac: {[1000000x1000000 double]  [1000000x1000000 double]
          [1000000x1000000 double]  [1000000x1000000 double]}
```

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

We can also initialize the variables with another initialization function:

```
[aAD, bAD, xAD, yAD] = initVariablesAD_diagonal(a, b, x, y);
```

which has the same interface but gives a different representations of AD variables, which we discuss in detail later on. We now have a `GenericAD` instance with a `DiagonalJacobian` that replaces the four sparse matrices:

```
disp(G)
```

```
GenericAD with properties:
  numVars: [4×1 double]
  offsets: [2×1 double]
   useMex: 0
      val: [1000000×1 double]
      jac: {[1×1 DiagonalJacobian]}
```

Computational efficiency: Figure 6.1 reports the execution time for each of the four operations in Listing 6.1 when the input variables are doubles or class objects of the sparse or diagonal AD type. With sparse AD data type of the standard AD library, MRST spends significantly more than 10 times as much time as with regular doubles. Computing **G** itself with derivatives takes 44 times as much time, but the performance penalty is significantly worse when we extract a subset of the larger vector. The new diagonal AD class, in contrast, significantly outperforms both the sparse AD and our goal of 10 times slower performance with four derivatives per

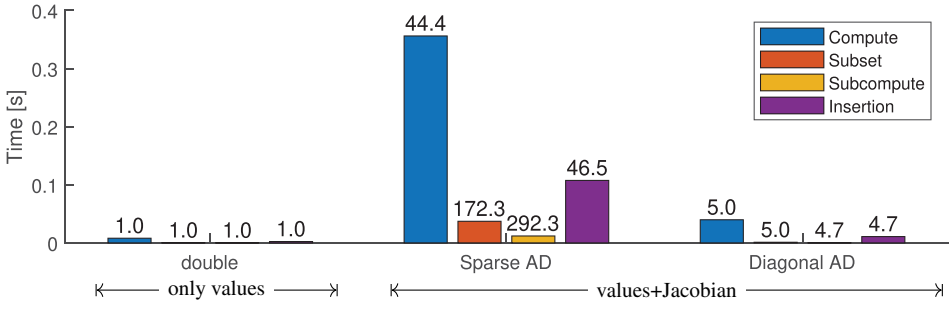

Figure 6.1 Time consumed by the four different operations in Listing 6.1 with vectors of 1 million cells. The times taken to evaluate values using doubles only are compared to the times consumed using two different versions of AD. The number above each bar indicates the relative increase in execution time when compared to computing values only using doubles.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

entry in $\mathbf{G}$. For all operations, the diagonal AD class uses roughly five times as much time as plain doubles. In the following, we try to answer two questions: Why does the standard AD in MRST use so much time? And what is done differently in the diagonal AD?

The answer lies in how the derivatives are represented. As you have already seen, the default AD implementation in MRST stores all derivatives as a list of sparse matrices. These are highly memory efficient for general sparse matrices because only nonzero entries are kept in memory. In reservoir simulation, you often want to manipulate the matrix blocks that correspond to individual independent variables in the full linearized system, and using a list of sparse matrices to represent the derivatives of the independent variable is much more efficient than using one big matrix for all of the derivatives, which is the standard choice in many forward AD libraries. However, the use of sparse matrices is not without drawbacks, because the bookkeeping of which matrix entries are present makes matrix operations more complex than for a dense matrix. In addition, MATLAB stores all sparse matrices in the column-major format [16], which means that all entries for a given column are stored together. Accessing all entries in a given column is therefore much more efficient than accessing all entries in a given row, because each entry then will be retrieved from a different column. Upon taking the subset of a AD vector, we must extract the sub-Jacobians that correspond to certain rows in the larger Jacobian, resulting in the construction of a new sparse matrix.

To improve execution speed, the diagonal AD uses a specialized class to store the Jacobians, which can be expanded to a standard sparse matrix when necessary. Because $\mathbf{G}_i$ only depends on values in the ith entries of each of the four inputs, most of the entries $\mathbf{J}_{ij} = \partial \mathbf{G}_i / \partial \mathbf{v}_j$ in the $L \times 4L$ Jacobian matrix will be zero. To be precise, $\mathbf{J}_{ij}$ can only be nonzero when j equals i, $i + L$, $i + 2L$, and $i + 3L$. We therefore say that $\mathbf{J}$ is *diagonal* in the sense that the `ADI` class would represent it as four sparse matrices in which only the diagonals are nonzero. Interpreted as one large sparse matrix, $\mathbf{J}$ has four nonzero bands. In general, the new diagonal AD class stores matrices in which the nonzero entries follow a simple pattern:

$$\mathbf{J}_{ij} = \begin{cases} v & \text{if } (j-1) \mod L = i - 1 \\ 0 & \text{otherwise.} \end{cases} \tag{6.5}$$

This trick enables highly efficient AD computations when operations are local to each cell. Subsection 6.2.4 discusses in more detail how this is done in practice, as well as how MRST uses diagonal-like representations even when the Jacobians are not as neatly structured as in (6.5).

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

COMPUTER EXERCISES

1. Here, **G** had 1 million entries, but you should test the example with larger or smaller `n` values. For smaller `n`, the sparse version may outperform the diagonal version; e.g., the author observed that the sparse version is comparable to the diagonal version when **G** has around 1 000 entries. How many elements do you need for the diagonal AD to outperform the sparse AD on your computer?
2. How does the difference between the AD types change with the number of threads? You can use `maxNumCompThreads` to control the number of threads. Experiment with the number of elements, threads, and AD variants to get a feel for the performance.

6.2.3 Sparse AD Backends in MRST

At the time of writing, MRST offers two different sparse AD backends. To compare and contrast the two, we use a simple flow example (`backendOptionsExample.m`) for which all of the existing backend options are available for experimentation.

Model setup: Consider a three-phase immiscible problem with pressure, water saturation, and gas saturation as primary variables. The specific details of the scenario are not important, because our primary concern is the structure of the Jacobians. We set up an initial state and initialize the model:

```
model.AutoDiffBackend = backend;           % Set backend
model  = model.validateModel();            % Validate model with new backend
state0 = initResSol(G, p0, s0);            % Set up initial state
state0 = model.validateState(state0);      % Validate initial state
state  = model.getStateAD(state0);         % Initialize AD-state
forces = model.getValidDrivingForces();    % Set up dummy forces
```

All changes in the following sections depend on the choice for the `backend` variable. Once everything has been set up, we can evaluate a few state functions via `getProp`:

```
% Get two primary variables, the mass in each cell and the phase flux
[p, sw] = model.getProps(state, 'pressure', 'sw');
[cm, v] = model.getProps(state, 'ComponentTotalMass', 'PhaseFlux');
eqs     = model.getModelEquations(state0, state, 1*day, forces);
```

For the discussion of the different Jacobian representations, it will be useful to make note of the dimensions of the grid in terms of cells and faces, which we verify from the dimensions of the outputs. Here, the grid has been set to 10×10 cells:

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

```
fprintf('Grid has %d cells with %d interfaces\n', numelValue(p), numelValue(v{1}))
```

```
Grid has 100 cells with 180 interfaces
```

The Standard Backend: `AutoDiffBackend`

The default backend is for all intents and purposes identical to the standard `ADI` class described in appendix A.5 of the MRST textbook [14]. If we examine the pressure primary variable, we see that the Jacobian consists of the usual list of sparse matrices:

```
disp(p)
```

```
  ADI with properties:
    val: [100×1 double]
    jac: {[100×100 double]  [100×100 double]  [100×100 double]}
```

Each sparse matrix in the cell array `p.jac` corresponds to the derivative of `p` with respect to the corresponding primary variable. This means that `p.jac{1}` represents $\partial\mathbf{p}/\partial\mathbf{p}$, `p.jac{2}` is $\partial\mathbf{p}/\partial\mathbf{S}_w$, and so on. Because `p` is itself a primary variable, the first entry is the identity matrix and the remaining entries are zero matrices.

Only the default backend provides instances of the `ADI` base class that all AD objects inherit from. This class implements the *canonical* representation that all AD functions should support. Other backends provide instances of the subclass `GenericAD`, which is flexible with respect to the Jacobian structure. If your code relies on particulars of the canonical per variable sparse Jacobian representation, you should use `assert(~isa(x, 'GenericAD'))` to throw an error inside your function if a nonstandard backend is in use.

The Sparse Backend: `SparseAutoDiffBackend`

The storage format and the underlying algorithms of the default `AutoDiffBackend` class may change in future releases of MRST and, hence, the sparse backend (`SparseAutoDiffBackend`) is introduced to enable backward compatibility and as a convenient way to ensure that a model can explicitly initialize the sparse backend. The general advantage of sparse backends is that Jacobians are easily manipulated as matrices and that any variable can depend on any other variable. Using a standard sparse matrix format ensures that the performance of this class

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

will automatically improve when MATLAB's and Octave's sparse implementations improve and that the underlying data elements can be directly passed to linear solvers that support column-major sparse matrices. The drawback is that certain sparse matrix operations like insertion and matrix–matrix operations may be expensive when in-memory data must be moved around to accommodate a new sparsity pattern. Typical simulations require a large number of linearizations, and the performance of the sparse representation is often a bottleneck if a fast linear solver is used.

Single-matrix representation: The sparse backend has an additional feature: We can change to a single-matrix representation to compute the full Jacobian as a sparse matrix, by setting `backend.useBlocks=false`:

```
disp(p)
```

```
GenericAD with properties:
  numVars: [3×1 double]
  offsets: []
   useMex: 0
      val: [100×1 double]
      jac: {[100×300 double]}
```

The `GenericAD` class now contains three data fields not present in the canonical `ADI` class, and the list of sparse matrices has been replaced by a single large Jacobian representing the entirety of $[\partial \mathbf{p}/\partial \mathbf{p}, \partial \mathbf{p}/\partial \mathbf{S}_w, \partial \mathbf{p}/\partial \mathbf{S}_g]$. Single-matrix storage will be faster for operations involving addition and multiplication of full matrices, because this enables better thread parallelization in MATLAB than operations that have to loop over lists of sparse submatrices. The disadvantage is that extracting a sub-Jacobian for one or more variables is more expensive for single-matrix storage, because a new sparse matrix must be created. The diagonals with respect to a group of variables are less accessible in the single-matrix backend, and functions that need to access groups of derivatives separately may not support this representation. One such example is the constrained pressure residual (CPR) linear solver, `CPRSolverAD`, which relies on access to each submatrix separately and does not currently support equations assembled with single-matrix AD as input.

6.2.4 High Performance: `DiagonalAutoDiffBackend`

The `DiagonalAutoDiffBackend` is designed to leverage dense linear algebra when possible and only uses the full sparse representation when strictly required. With many options that improve execution speed, this is the primary test bed for AD performance improvements in MRST.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Optimized Representation of Diagonal Matrices

To understand how the class is constructed to optimize storage and number of flops, we again examine the initialized AD objects for the `p` variable after switching backends:

```
disp(p)
```

```
  GenericAD with properties:
    numVars: [3×1 double]
    offsets: [2×1 double]
     useMex: 0
        val: [100×1 double]
        jac: {[1×1 DiagonalJacobian]}
```

Here, we see that the list of sparse matrices that represented the Jacobian in the `ADI` class has been replaced by an instance of the `DiagonalJacobian` class:

```
Jp = p.jac{1}; Jw = sw.jac{1}; % Get pressure and S_{w} Jacobians
disp(Jp)                       % Show the pressure Jacobian
```

```
  DiagonalJacobian with properties:
                 diagonal: [100×3 double]
                      dim: [100 3]
                   subset: []
                        :
```

The `dim` property indicates that this is a Jacobian with respect to a primary variable vector with 100 elements and three variables per element. The `diagonal` property stores the local derivatives for each element as an $n_v \times n_d$ dense matrix, where n_v is the number of values and n_d is the number of local derivatives for each value. The first column of the Jacobian is all ones, whereas the remaining two columns are all zero. These zeros are stored in memory, however, which means that, e.g., multiplying `p` with a constant would result in additional multiplication operations when compared to the sparse representation. The values stored in memory are shown in Figure 6.2 for the three storage formats we have encountered so far: separate sparse blocks, a single large sparse block, and the diagonal Jacobian.

It is easy to think that the explicit storage of all diagonal entries would mean reduced performance when many entries have zero value, but the overhead of tracking the sparsity far exceeds the extra time spent multiplying out extraneous zeros: With a grid of dimensions $1\,000 \times 1\,000$, the author recorded a speedup of 2.4 by switching to the diagonal backend for the operation `x = p*5`.

Why is it more efficient to store the derivatives as a dense matrix? If we multiply vectors $\mathbf{x}$ and $\mathbf{y}$ element-wise with the corresponding Jacobians $\mathbf{J}_x$ and $\mathbf{J}_y$, a vector-valued variant of (6.4) reads

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

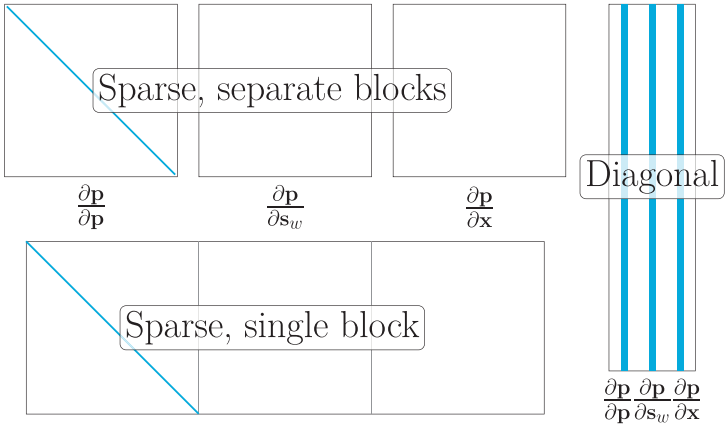

Figure 6.2 Three different ways of storing the Jacobian of the same primary variable in MRST: Sub-Jacobian blocks that distinguish between each type of primary variables (top), a single large sparse matrix (bottom), or storing the diagonals as a dense matrix (right). The bands here are not the nonzero elements; rather, they are values that are explicitly stored in memory. For the sparse representations, this obviously coincides with the nonzero entries.

$$\mathbf{J}_{xy} = \texttt{diag}(\mathbf{y})\mathbf{J}_x + \texttt{diag}(\mathbf{x})\mathbf{J}_y. \tag{6.6}$$

Here, `diag()` takes a vector and produces a matrix with the vector entries on the diagonal. The effect of this is that every row of the Jacobian $\mathbf{J}_y$ is multiplied by the corresponding value of $\mathbf{x}$ and vice versa. To understand how the diagonal backend gets better performance, we can manually set up and examine the operations each backend performs when realizing (6.6). Letting `m` be the number of elements and `nder` the number of derivatives per element, we initialize:

```
Dx = rand(m, nder);  Dy = rand(m, nder);  % Diagonals of Jacobians
x  = rand(m, 1);     y  = rand(m, 1);     % Vector values
```

Let us first consider how operation (6.6) would be computed with the sparse backend. We first initialize sparse matrices to represent each of the sub-Jacobians:

```
Jx = cell(1, nder); Jy = cell(1, nder);
for i = 1:nder
    Jx{i} = sparse(1:m, 1:m, Dx(:, i), m, m);  % m-by-m diagonal matrix, J_xi
    Jy{i} = sparse(1:m, 1:m, Dy(:, i), m, m);  % m-by-m diagonal matrix, J_yi
end
```

Computing the full Jacobian itself consist of creating two sparse matrices that contain $\mathbf{x}$ and $\mathbf{y}$ on their respective diagonals and multiplying and adding each diagonal:

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

```
J_sparse = cell(1, nder);          % Storage
dx = sparse(1:m, 1:m, x, m, m);    % diag(x)
dy = sparse(1:m, 1:m, x, m, m);    % diag(y)
for i = 1:nder, J_sparse{i} = dx*Jy{i} + dy*Jx{i}; end
```

For a diagonal matrix, the Jacobians are represented as the rectangular matrices `Dx` and `Dy` and we can directly evaluate the Jacobian[2]:

```
J_diag = Dx.*y + Dy.*x;
```

To compare, we perform each operation 100 times and output the average time. For five derivatives and 1 million elements:

```
Sparse: 0.2972s. Diagonal: 0.0076s (39.3 speedup from diagonal)
```

This amounts to nearly 40 times speedup by using the diagonal representation. One possibility would be that the `for` loop required for the sparse representation has a high cost. If we consider a single derivative per element to avoid the `for` loop,

```
Sparse: 0.1100s. Diagonal: 0.0018s (62.0 speedup from diagonal)
```

we see that the gap in performance is even larger. Modern CPUs have exceptional performance when performing the same operation on two arrays that lie contiguously in memory, and even the most optimal sparse implementations thus have a hard time achieving comparable performance to dense linear algebra.

The `subset` property of the Jacobian makes it possible to retain efficiency on subsets of diagonal Jacobians. For instance, slicing rows from a diagonal Jacobian, which occurs when extracting a subset of a AD vector, also gives a diagonal:

```
Jps = Jp(5, :) % Pick the fifth element of pressure Jacobian
```

```
DiagonalJacobian with properties:
               diagonal: [1 0 0]
                    dim: [100 3]
                 subset: 5
                      :
```

Note that the `dim` field is still `[100,3]`, because the subset is taken from a primary variable with 100 entries and three variables per entry. If we were to represent this

[2] On MATLAB versions prior to R2016b, we use `bsxfun` instead of the implicit expansion: `J_diag = bsxfun(@times, D1, v2)+ bsxfun(@times, D2, v1)`. MRST automatically determines whether the implicit expansion is available and uses the fastest version available.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

subset as a sparse matrix, it would be a 1×300 matrix. The increased speed of the diagonal structure is maintained whenever possible; e.g., when a modified subset is reinserted into the same position:

```
Jp(5, :) = 2*Jps;      % Inserting in the same position still gives a diagonal
class(Jp)              % -> DiagonalJacobian: Diagonal structure preserved
Jp(6, :) = 3*Jps;      % Insertion in another position results in sparse
class(Jp)              % subset mismatch -> double: Sparse matrix
```

Operations on subsets get the efficiency of the diagonal representation if possible. If an operation requires a sparse representation, the object is automatically converted:

```
Jws = Jw(5, :);        % Take the fifth element of S_w Jacobian
class(Jps + Jws)       % Different variables, same subset -> DiagonalJacobian
class(Jps + Jw(6, :))  % Different variables, different subset -> double
```

In addition to the automatic conversion, we can expand an instance of the diagonal representation to a matrix at any time by explicitly casting to sparse:

```
Jp = sparse(Jp);
fprintf('Jacobian has type %s with dimensions %d by %d.\n', class(Jp), size(Jp))
```

```
Jacobian has type double with dimensions 100 by 300.
```

In this way, the diagonal backend can still represent fully general derivatives when needed, although the disadvantages with sparse matrices will then apply to any subsequent operations on the Jacobians. In Subsection 6.2.4, you will see how we can retain the diagonal efficiency even when working with nondiagonal values; for instance, when we have values on faces that are differentiated with respect to cell variables on an unstructured grid.

Finally, we note that the diagonal backend will group the derivatives of vectors of the same length together. In the following case, it will store the Jacobians as two separate groups:

```
u = zeros(10, 1); e = zeros(5, 1);
[a, b, c, d] = initVariablesAD_diagonal(u, u, e, e);
disp(a)
```

```
  GenericAD with properties:
        :
        val: [10×1 double]
        jac: {[1×1 DiagonalJacobian]  [1×1 DiagonalJacobian]}
```

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

The first Jacobian has derivatives with respect to the two vectors with 10 entries each and the second Jacobian represents the derivatives with respect to the two primary variables with five entries each:

```
disp(a.jac{1})
```

```
DiagonalJacobian with properties:
         diagonal: [10×2 double]
           subset: []
              dim: [10 2]
                :
```

```
disp(a.jac{2})
```

```
DiagonalJacobian with properties:
         diagonal: [10×0 double]
           subset: [10×1 double]
              dim: [5 2]
                :
```

Because we here examined the first primary variable, all derivatives in the second group, which correspond to the second primary variable, are zero. This leads to two observations: (i) the diagonal has zero for the second dimension to represent a zero Jacobian and (ii) the subset is already initialized. The subset entries are all zero values, which indicates that any value can be inserted anywhere in the array without switching to a sparse matrix.

Modified Discrete Operators

The diagonal Jacobians we have described so far will turn into sparse matrices when they are multiplied by a general matrix. Many of the discrete operators used in the assembly of the linearized system for (6.1) are linear maps that take cell values as input and produce results on the faces; i.e., a mapping of the type $\mathbb{R}^{n_c} \to \mathbb{R}^{n_f}$. For example, the discrete gradient and divergence operators in MRST are implemented using a sparse matrix $\mathbf{D} \in \mathbb{R}^{n_c} \times \mathbb{R}^{n_f}$:

$$\texttt{grad}(\mathbf{p}) = -\mathbf{D}\mathbf{p}, \quad \texttt{div}(\mathbf{v}) = \mathbf{D}^T\mathbf{v}. \tag{6.7}$$

As described in [14, subsection 4.4.2], each row of this matrix corresponds to a face and contains two nonzero entries in the positions where the cells belonging to that face are found. The same principle applies to the other operators that compute face values from cell values, namely, `faceUpstr` and `faceAvg`. These are all used to compute discrete phase fluxes of the form

$$\mathbf{v}_\alpha = -\texttt{upw}(\boldsymbol{\lambda}_\alpha)\mathbf{T}_f\left(\texttt{grad}(\mathbf{p}_\alpha) + g\,\texttt{favg}(\boldsymbol{\rho}_\alpha)\texttt{grad}(\mathbf{z})\right). \tag{6.8}$$

We saw earlier that computing derivatives for these multiplication and addition operators is significantly more efficient if we can exploit the structure of the Jacobians to avoid introducing unnecessary intermediate sparse matrices. The diagonal backend has an option to replace the existing matrix-based operators with custom operators tailored to expressions involving diagonal Jacobians. This setting

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

is enabled by default but can be disabled by setting the public backend property `modifyOperators` to `false`.

To see this effect, we examine the Jacobian of one of the phase fluxes we retrieved earlier. When modified operators are not enabled, we get a sparse Jacobian of 180×300 entries, but if the option is enabled, we get a class instance instead:

```
vw = v{1} % Get the water flux
```

```
>> backend.modifyOperators=false;
 GenericAD with properties:
       :
   val: [180×1 double]
   jac: {[180×300 double]}
```

```
>> backend.modifyOperators=true;
 GenericAD with properties:
       :
   val: [180×1 double]
   jac: {[1×1 FixedWidthJacobian]}
```

The new class, `FixedWidthJacobian`, is designed to represent Jacobians of operations on multivariate functions when each entry in the output vectors depends on *a fixed number of entries in the original vector*. If modified operations are activated in the backend, face averages, upwinded quantities, and gradients all get Jacobians of this class. Element-wise operations on the output from these discrete operators also retain the same structure, which means that (6.8) will also have a dense Jacobian representation. This means that the intermediate multiplications and additions are all able to get the efficiency through the same means as the regular diagonal Jacobians.

The Jacobian of the discrete water flux from our example gives several hints to how this is implemented in practice:

```
disp(vw.jac{1})
```

```
  FixedWidthJacobian with properties:
                 map: [180×2 double]
             mapName: 'interiorfaces'
        parentSubset: []
            diagonal: [180×6 double]
                 dim: [100 3]
              subset: []
                   :
```

Because the class is derived from the regular `DiagonalJacobian` class, we still have the `diagonal` property. Though it is expected that we have one entry per face, we also observe that there are six columns instead of the three we saw for the cell pressure in the beginning of this section. Each face separates two cells, which means that the Jacobian will have dimensions $n_f \times 2n_d$. Figure 6.3 outlines the layout of the diagonal matrix for face quantities for a small grid. Here, the neighborship matrix N is stored as the `map` property. In the case of face values, the rows tell us the pair of cells any given face value depends

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

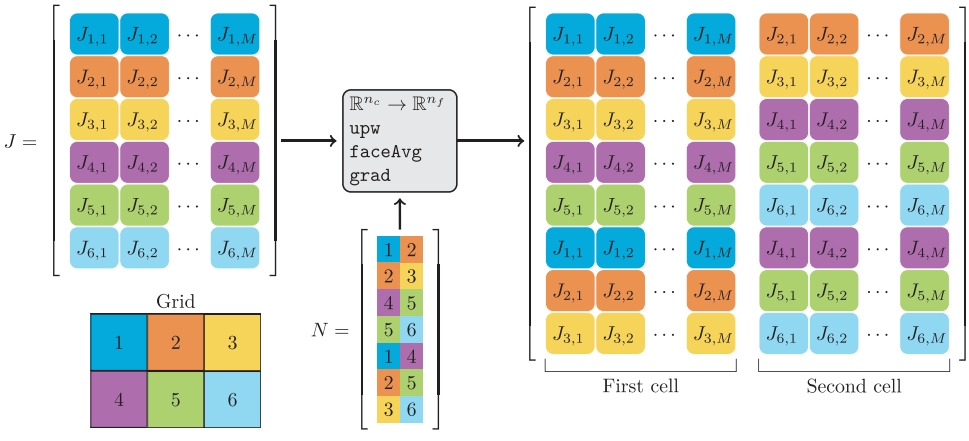

Figure 6.3 The in-memory diagonal Jacobian structure of derivatives local to each cell (upper left) gets converted into a Jacobian defined on interior faces (right) for a grid with two cells per faces (lower left). This way, the Jacobian can still be represented as a dense matrix.

on. The property `mapName` can contain a string that uniquely defines `map`, here `'interiorfaces'`. In this way, the class does not need to check each entry of `map` to see whether two fixed-width Jacobians can be safely multiplied together. As with the regular `DiagonalJacobian` class, we can seamlessly operate on subsets of variables and expand to a sparse matrix when encountering, e.g., a general matrix–vector product. We also have a `parentSubset` that may contain the subset array of the original cell values used to produce the fixed-width Jacobian.

The fixed-width representation reuses most operations directly from the regular diagonal Jacobian class: The class consists of mere 130 lines, most of which concern conversions to sparse matrices and equality comparisons between two Jacobian classes of different type. The backend also includes a custom version of `div` and `AccDiv`. These operators take face quantities as input and produce output in each cell, and the custom versions can exploit the fixed-width representation for faster assembly. At this stage, the sparse Jacobian is usually assembled, unless a specific option is set, as detailed in Subsection 6.2.4. As with the face operators, these custom operators degrade gracefully to the standard sparse versions when provided with sparse Jacobians.

Row-Major Option

MATLAB stores all matrices in column-major order. This means that accessing consecutive entries in a single column is more efficient than accessing the same

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

number of consecutive entries in any given column. For example, the following two codes produce identical outputs with the same number of flops:

```
for i = 1:nrow      % Row outer
    for j = 1:ncol % Column inner
        A(i, j) = i + j;
    end
end
```

```
for j = 1:ncol      % Column outer
    for i = 1:nrow % Row inner
        A(i, j) = i + j;
    end
end
```

With `nrow` and `ncol` both set to 2 000, the second alternative is almost 2.4 times faster than the first. For historical reasons, MRST stores solution quantities such as pressures or saturations as column vectors, and the default behavior of the diagonal backend matches this: Each row of an $n_v \times n_d$ `diagonal` matrix corresponds to all derivatives of a single value. Unfortunately, this can be inefficient if all derivatives of a value are retrieved more often than, e.g., getting the first derivative of all values. With the `rowMajor` option set, the backend stores the *transpose* of (6.5) in each object to improve memory locality. Figure 6.4 demonstrates the difference between the two representations when accessing the data consecutively in memory.

MEX Acceleration

The results in Figure 6.1 were achieved with pure MATLAB code and show that users without access to a C++ compiler can also benefit from switch-

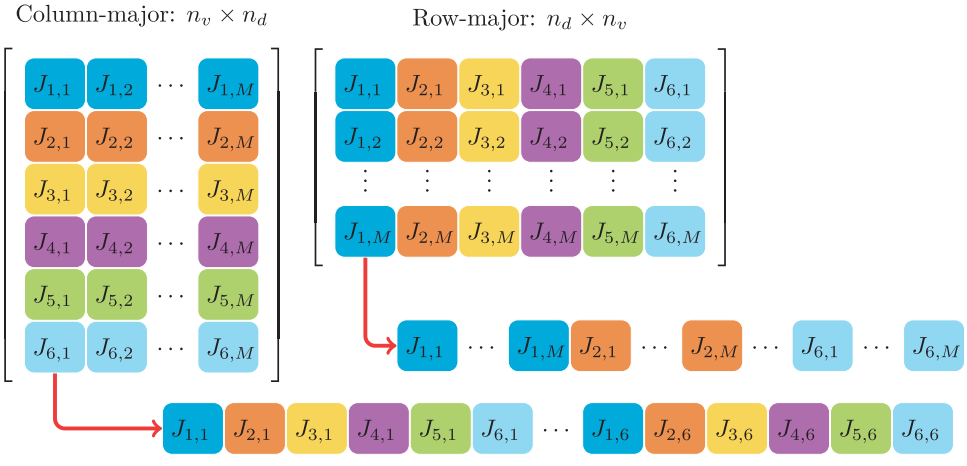

Figure 6.4 The diagonal backend can store the diagonals containing nonzero derivatives as either column-major or row-major. If using a row-major ordering, the derivatives for each value are consecutive in memory, making retrieval of all derivatives much more efficient.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Table 6.2 *Overview of MEX-accelerated operators for the diagonal AD backend. Some operators have acceleration for both calculating values and computing the associated Jacobian.*

Operation	Function	MEX Value	MEX Jacobian
`upw`	`singlePointUpwind`	Yes	Yes
`grad`	`twoPointGradient`	Yes	Yes
`favg`	`faceAverage`	Yes	Yes
`div`	`discreteDivergence`	Yes	Yes
$\texttt{diag}(\mathbf{y})J_x$	`diagMult`	–	Yes
$\texttt{diag}(\mathbf{y})J_x + \texttt{diag}(\mathbf{x})J_y$	`diagProductMult`	–	Yes
Diagonal to sparse	`sparse`	–	Yes

ing to the diagonal backend. The runtimes can be further reduced by setting `backend.useMex=true` and thereby enable MEX acceleration of many operations in the AD library. Table 6.2 list operators that have accelerated versions available. The accelerated versions are written in C++ and use OpenMP for parallelization, enabling MRST to take advantage of multiple threads without any additional toolboxes.

Deferred Assembly Option

Normally, the `AccDiv` and `Div` operators lead to the sparse matrix assembly. The diagonal backend also has an option of deferred assembly, so that these operators instead return an intermediate representation that can then be assembled into other matrix formats; e.g., when using a linear solver that prefers another input than the default compressed sparse column format. Examining the water conservation equation, we see the difference by changing the option:

```
ew = eqs{1} % Equation for conservation of water component
```

```
>> backend.deferredAssembly=false;
GenericAD with properties:
  val: [100×1 double]
  jac: {[100×300 double]}
    :
```

```
>> backend.deferredAssembly=true;
GenericAD with properties:
  val: [100×1 double]
  jac: {[1×1 ConservationLawJacobian]}
    :
```

With deferred assembly enabled, the Jacobian contains the accumulation terms and the discrete flux as diagonal and fixed-width Jacobians, respectively:

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

```
disp(ew.jac{1})
```

```
  ConservationLawJacobian with properties:
                     flux: [1×1 FixedWidthJacobian]
             accumulation: [1×1 DiagonalJacobian]
       divergenceOptions: [1×1 struct]
```

This class can then add cell-wise contributions to the accumulation term. As before, we can always cast this class to sparse to get the scalar compressed sparse column representation or use the intermediate Jacobians to assemble the matrix into another type. In Example 6.5.2, this form is used to assemble a block CSR matrix that is passed directly onto a linear solver.

A Few Words of Caution

The advantage of the diagonal representation is that it can often be more efficient than the default sparse implementation. In the end, the diagonal representations will gracefully degrade to sparse matrices as needed, but if this occurs early in a complex expression, the performance benefits may be lost. If your code uses additional discrete operators beyond those included with MRST, you may have to implement a diagonal version that uses the fixed-width Jacobian if a sparse representation of the discrete operator is inefficient. This part of MRST is relatively new and also under active development, but the principles discussed are general and should be relevant to anybody interested in reducing computational overhead. We cannot yet guarantee that there will not be bugs in untested corner cases and thus recommend that you first validate your implementation with the canonical sparse AD backend and then only start experimenting with the additional options when you progress your research to larger cases and assembly time becomes significant.

6.2.5 Performance of AD Backends

We end the discussion by presenting three test cases that compare and contrast the performance and scalability of the AD backends currently available on larger and more representative setups, sampled from multiphase simulation models.

Benchmarking Operations for Different Backends

You have already seen that using the diagonal backend improves certain element-wise operations. MRST includes the routine `benchmarkAutoDiffBackends`, which performs a more systematic performance test for any given set of backends. The use of this routine is demonstrated in `exampleBenchmarkBackends`, in which different backends can be compared for grids of varying size and with a varying number of degrees of freedom. Here, we only report one such case

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

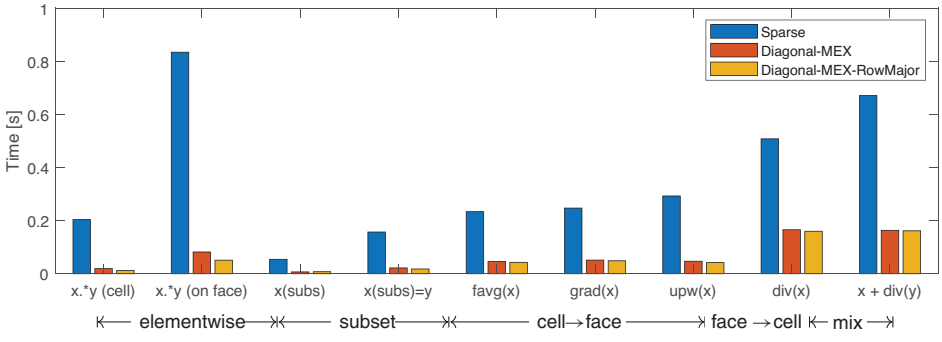

Figure 6.5 A benchmark of three different backends for automatic differentiation with a million cells and five primary variables in each cell. The first tests are element-wise products, followed by subset operations, cell-to-face operators, and finally a face-to-cell operator and a mixed operator.

with 1 million cells and five primary variables per cell, but we encourage you to experiment with the different options to see what parameters give the best performance on your configuration. We first define the backends we would like to benchmark:

```
sparseBlocks = SparseAutoDiffBackend();
diagRowMex   = DiagonalAutoDiffBackend('useMex', true, 'rowMajor', true);
diagColMex   = DiagonalAutoDiffBackend('useMex', true, 'rowMajor', false);
backends     = {sparseBlocks, diagRowMex, diagColMex};
```

We then call the benchmark itself, passing the Cartesian dimensions of a desired test grid as the first input. Alternatively, we could either pass a model or a grid to benchmark on a specific case:

```
dim = [100, 100, 100]; % Get a 100 by 100 by 100  Cartesian grid
results = benchmarkAutoDiffBackends(dim, backends, 'block_size', 5);
```

The first backend is used as the *reference* when measuring speed and correctness of the backends. Each operation is performed a number of times based on the `'iterations'` optional parameter and the average wall time is output together with the observed speedup if no outputs are requested or the `'verbose'` option is enabled. The output contains a large number of tests, which are provided in Appendix B.

Figure 6.5 plots a few element-wise operations, operations on 20% of the elements, as well as some discrete operators. By using the most efficient diagonal backend with row-major storage of Jacobians and MEX acceleration, we obtain a speedup in the range of 5 to 10 for element-wise operations and a factor of

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

three to five for the discrete operators. The performance can be accelerated up to three orders of magnitude in the case of smaller subsets, in line with our earlier observations.

Computer exercises

1. We only tested three backends. Change the example so that it includes the single-block sparse representation and any other options you would like to test.
2. How do the MEX backends perform on your machine? Are there any differences compared to our reported results?
3. The operators can be tested on any kind of grid. How do they perform on a corner-point grid having a variable number of faces per cell? Try to set up one of the grids from the MRST textbook and replace the `dims` input with this grid.

Assembly Benchmark: Different Sets of Governing Equations

In the preceding section, we assessed the performance of MRST's AD library on many of the key discrete operations found in a reservoir simulator. The timing of each operation may be difficult to relate to the full assembly process in a simulation model, because some operations may occur more often than others in a nonlinear problem. To assess the performance of MRST's assembly in practice, we consider the linearization of a parameterized test problem. Just as for the individual AD operations, MRST has a dedicated routine for benchmarking assembly of a single linearized system; `assemblyBenchmarkAD` can be used to estimate the assembly speed for a variety of different models:

```
results = assemblyBenchmarkAD(N, backend, physics, 'wells')
```

The test problem is posed on a Cartesian grid with $N \times N \times N$ cells. If the fourth argument is `'wells'`, the assembly includes a set of four vertical wells, placed in the corners of the domain and perforated throughout all N layers of the model. We let N vary from 20 to 126, with the smallest grid having 8 000 cells and the largest two million cells. In the example `showADBenchmarkAssembly`, three different sets of governing equations are considered:

1. Three-phase immiscible flow: three degrees of freedom per cell, no capillary pressure or gravity, and linear relative permeabilities.
2. Three-phase black oil with the SPE 9 benchmark fluid [10]: three degrees of freedom per cell, capillary pressure, gravity, dissolved gas, no vaporized oil.
3. Liquid–vapor compositional problem with overall composition formulation: The fluid model is taken from the SPE 5 benchmark [11], with six component degrees of freedom per cell. The compositional model allows components to be present in both phases.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Table 6.3 *Overview of the execution time in seconds for a single linearization using the diagonal AD backend with C++ acceleration for three different fully implicit flow systems posed on a Cartesian grid with $N \times N \times N$ cells. See Figure 6.6 for a plot of the same data.*

Grid		Single-phase		Immiscible		Black oil		Compositional	
N	# Cells	Base	Wells	Base	Wells	Base	Wells	Base	Wells
20	8 000	0.02	0.04	0.04	0.06	0.07	0.10	0.44	0.54
25	15 625	0.02	0.03	0.04	0.07	0.09	0.12	0.63	0.74
30	27 000	0.03	0.04	0.05	0.08	0.10	0.14	0.90	1.05
35	42 875	0.03	0.04	0.07	0.10	0.14	0.19	1.21	1.36
50	125 000	0.04	0.07	0.13	0.20	0.28	0.38	2.79	3.09
75	421 875	0.10	0.14	0.36	0.52	0.85	1.09	9.26	9.95
100	1 000 000	0.21	0.28	0.78	1.11	1.87	2.37	20.57	22.13
126	2 000 376	0.39	0.54	1.53	2.25	3.77	4.62	40.89	44.17

We consider the assembly of a single linearized set of equations. For black oil and compositional, we let 10% of the domain be in the two-phase hydrocarbon state, so that cells contain both free gas and oil–liquid, because this leads to additional work during assembly for these models. The overall compositional solver performs a number of linearizations to get derivatives of the outputs from the flash equations with respect to the chosen primary variables in the region with both liquid and vapor present; see Chapter 8 for more details.

In the following, we focus on the diagonal representation with C++ acceleration, which is the fastest implementation currently available. The results are shown in Table 6.3. We note that each combination of grid and fluid system has two benchmarks in the table. If the problem has wells, we can describe the linearized system by dividing the Jacobian into four parts:

$$J = \begin{bmatrix} J_{rr} & J_{rw} \\ J_{wr} & J_{ww} \end{bmatrix}. \tag{6.9}$$

In the upper row, J_{rr} represents the Jacobian of the reservoir equations with respect to primary variables defined in reservoir cells and J_{rw} is the derivative of the same equations with respect to the well primary variables. In the lower row, the well equations are differentiated with respect to reservoir primary variables, J_{rw}, and the well variables themselves, J_{ww}. In the table, "base" refers to the assembly of J_{rr} alone, whereas the "wells" results correspond to the coupled system that contains all four blocks of J.

Because the length of the wells grows with the vertical extent of the domain, the overhead of assembling the well equations remains roughly the same percentage of

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

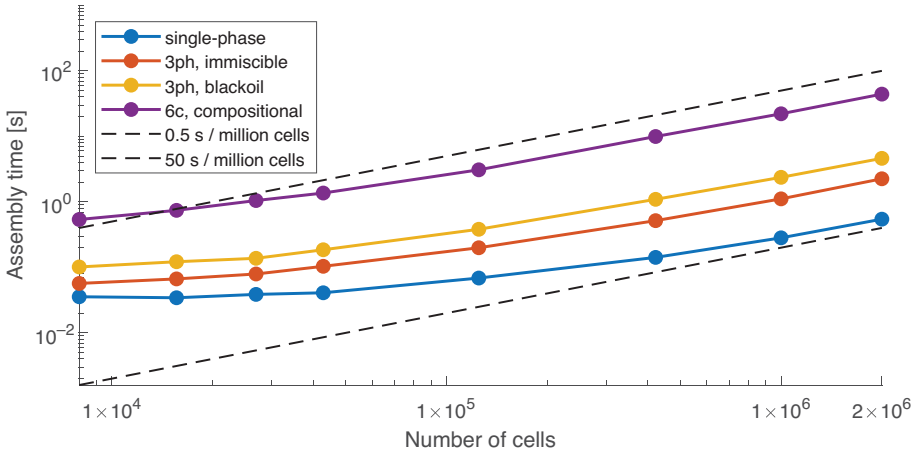

Figure 6.6 The time consumed by the diagonal AD backend with C++ acceleration to assemble a single linearized residual with respect to all primary variables for three different fluid models: immiscible three-phase, three-phase black oil, and compositional. All models contain five wells perforated in all layers of the domain. Note that the y axis is logarithmic, with two different types of linear scaling shown as black guards that correspond to 0.5 seconds per million cells and 50 seconds per million cells. The data for this plot are provided in Table 6.3.

the total time as the grid resolution increases. The different physical systems have widely different costs: Whereas a million-cell scalar problem can be assembled in 0.28 seconds with wells, a million-cell compositional system with the same wells takes 44.17 seconds. For the latter, the large number of interactions between the component pairs in the equation of state and the calculation of implicit derivatives consume significant time. Assembly of the conservation equations takes approximately 40% of the time for the compositional model and closer to 95% for the other models. The immiscible and black-oil systems are less expensive, clocking in at assembly times of 1.11 and 2.37 seconds, respectively, for a million cells with wells.

The results for assembly with wells are plotted in Figure 6.6. On the smallest grids, the assembly cost is dominated by computational overhead of the classes and functions in MATLAB and hence does not increase significantly with the number of degrees of freedom. The state functions used by the simulator to compute the residual equations are all vectorized over all cells, so that the number of function calls is the same for 100 cells and for 1 million cells. For this reason, we observe a linear trend once the vector operations dominate the constant overhead.

Assembly Benchmark: Parallel Performance

Developing highly parallel programs is fraught with difficulties, and benchmarking of parallel programs even more so. It is generally accepted that codes must be

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

(re)written with parallel performance in mind from the ground up to benefit from a large number of processors. The MEX-accelerated AD backends do not aim to make MRST's prototyping simulators massively parallel. Rather, the goal is that each of the individual AD routines should be *shared memory parallel* to get reasonable performance on modern workstations that have many cores, without sacrificing any of the prototyping flexibility.

In the context of strong scaling, in which a large problem is executed faster by adding more threads, it is natural to compare to Amdahl's law. If we define the *serial* fraction of a program as F_s, so that the part of the program that can fully take advantage of any number of processors is $1 - F_s$, the speedup achieved when going from 1 to N_p processors has a closed-form expression [1]:

$$S_s = \frac{1}{F_s + (1 - F_s)/N_p}. \tag{6.10}$$

We can exploit Amdahl's law to estimate the fraction of the assembly that is parallel without having to test individual parts of the simulator. To exemplify, we repeat the immiscible assembly from the previous example with a varying number of threads active for a model with fixed size (source code: `parallelScalingAD`). The test was run on a dual-CPU workstation with two Intel Xeon E5-2630 CPUs with 2.6 GHz base speed and 128 GB of RAM with 12 cores in total. The Intel Turbo Boost feature was disabled to ensure that the single-thread performance is close to 1/12th of the total capacity of the processor.

If we perform the test for a model with 6 million cells, so that each thread has at least 500 000 cells, MRST uses 23.9 seconds in total for a single thread and 3.8 seconds for 12 threads, a total speed up of 6.3. Figure 6.7 reports the results for 1 to 12 threads, together with the theoretical scaling according to (6.10) for a serial fraction F_s of 0.1 and 0.05, as well as the idealized case with speedup

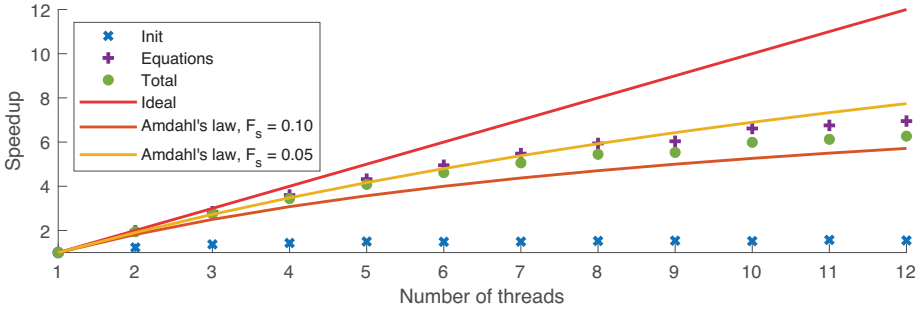

Figure 6.7 Strong scaling of an immiscible three-phase assembly case with a total of 6 000 000 cells and up to 12 threads.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

proportional to the number of threads. Judging from the figure, the total assembly would be classified as between 90% and 95% parallel according to the estimate from Amdahl's law, so that the assembly of the equations themselves with AD is largely parallel. The initialization of AD variables and state, although small for the single-thread case, does not take advantage of more than a few threads.

Common Pitfalls with AD

We end the performance discussion by pointing out a few potential pitfalls. The first comes from the way MATLAB is designed:

> MATLAB, and by extension Octave, does not perform type conversion when assignment by indexing (`subasgn`) is used. This can sometimes result in less than intuitive behavior when preallocating arrays as doubles.

We can easily make an example illustrating the dangers of implicit conversion. Note that as of MRST 2019a, this example will result in a runtime warning from the AD implementation:

```
x = zeros(10, 1);        % Initialized vector
y = initVariablesADI(1); % Make AD
x(5) = y;                % Insert AD object in array. Will produce a warning.
disp(x)                  % We still have a double! Derivatives were lost.
```

For this reason, the backend can also convert values of type `double` to AD variables with zero-initialized derivatives, which is useful when preallocating storage for vectors, as the previous code excerpt illustrated. Let us consider a function `myfun`, which has many numerical input arguments. We do not know which of the numerical arguments are AD (if any), but we can still write an AD-capable function:

```
function z = myfun(model, x, y)
    s = getSampleAD(x, y);               % Find whichever of x and y are AD
    z = zeros(10, 1);                    % Class of double
    z = model.backend.convertToAD(z, s); % Convert to AD with same context as s
    % ... do operations on z as normal
end
```

Another important consideration is that the performance of an operation decreases with the number of derivatives and their storage format. You can often obtain better performance by rearranging operations to minimize the number of operations on nondiagonal Jacobians.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

6.3 High-Performance Linear Solvers

The runtime of a typical MRST simulation can be classified into three parts: (i) the assembly and update functions, primarily governed by the performance of AD and the state functions in use; (ii) the time spent in the linear solver; and (iii) the number of nonlinear iterations required to meet the convergence criteria.

We have already explained how you can reduce the time spent in assembly by replacing the AD backend. Next, we consider the linear solver, with a focus on the possibilities that lie in the compiled solvers you can automatically download and use with MRST. We will not go into the same level of detail for the nonlinear solvers but instead refer you to [14, subsection 12.3.2]. Our starting point is that you have a simulation that convergences nicely after tweaking the residual tolerances and nonlinear solver parameters like acceptance factor, relaxation, line search, and maximum updates for pressure and saturation, but the simulation is still too slow.

The helper utility `getNonLinearSolver` has several options for setting up a nonlinear solver with automatic timestepping and a suitable linear solver. These reasonable conservative defaults are used throughout MRST, including the setup routines that use input files described in Subsection 6.4.1.

6.3.1 Selecting Different Linear Solvers

Whereas configuring the nonlinear solver is important to get good computational performance, the *linear* solver is usually the factor that limits the size of the problems you can solve with MRST. Unfortunately, the one-size-fits-all linear solver is yet to be invented and therefore there are a number of linear solvers available in the AD-OO framework. These are all derived from the base class `LinearSolverAD`. If you have instantiated one such class object, `linsolver`, there are basically two ways you can pass it to your simulator. The first option is to configure the `NonLinearSolver` object, which is responsible for calling the linear solver and thus always contains a linear solver class object:

```
nls.LinearSolver = linsolver;
[ws, states, report] = ...
    simulateScheduleAD(state0, model, schedule, 'NonLinearSolver', nls)
```

Alternatively, you can pass the `linsolver` object directly to the simulator if you are otherwise satisfied with the default nonlinear solver:

```
[ws, states, report] = ...
    simulateScheduleAD(state0, model, schedule, 'LinearSolver', linsolver)
```

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Overview of Linear Solvers in the AD-OO Framework

To start, we complement the discussion in subsection 12.3.4 of the MRST textbook [14] by listing all of the linear solvers available to MRST, together with their requirements. First of all, the software offers a number of self-contained linear solvers that can be used out of the box:

- `BackslashSolverAD`: This default option corresponds to the `\` (`mldivide`) operator. Depending on the structure of the input matrix, different direct solvers will be used. Many of the linear systems produced by simulation models in MRST are non-Hermitian and will normally be solved by LU factorization, which is fairly expensive for models with more than a few thousand degrees of freedom. The limited scaling of direct factorization for sparse systems is usually the factor that prevents you from simulating larger cases with MRST.
- `CPRSolverAD`: An implementation of different CPR [4, 24] preconditioners for fully implicit systems. The solver creates an approximate pressure equation, which is passed onto another, user-configurable linear solver of any type. It then uses MATLAB's built-in scalar incomplete LU factorization ILU(0) on the entire system.
- `GMRES_ILUSolverAD`: A pure MATLAB implementation of GMRES, preconditioned with scalar ILU(0). Can often perform better than the direct solvers, especially for pure transport systems.
- `HandleLinearSolverAD`: Not technically a complete linear solver but rather a *wrapper* class that takes a function handle on the form `@(A, b)` as input. Useful for quickly integrating your favorite linear solver, if it already has a MATLAB interface, into an MRST simulation without writing any code.

Because the first three of these base solvers primarily rely on standard MATLAB functionality, their computational performance is somewhat limited.

Large sparse systems arising from the discretization of partial differential equations are usually best solved with a combination of different sparse preconditioning techniques. To account for this, we have recently introduced a number of interfaces to external linear solvers in MRST, including both general iterative solvers and algebraic multigrid (AMG). The latter is especially useful for solving elliptic-like equations such as pressure or steady-state thermal distributions, either as a stand-alone system or for the elliptic part of the two-stage preconditioner in `CPRSolverAD`. These external solvers can be used if the prerequisite dependencies are met, or their interfaces can serve as the base for developing links to your favorite linear solver.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

- `AGMGSolverAD`: Agglomeration-based AMG, wrapping the `agmg` solver [20]. Precompiled binaries are available from the AGMG website [19] at no cost for academic use. A license is required for commercial use. Over the years, we have used AGMG extensively with the incompressible family of solvers in MRST but less so for solvers based on the AD-OO framework.
- `AMGCLSolverAD`: The AMGCL library is a header-only C++ library for solving sparse linear systems developed by Demidov [7]. The source code is released under the permissive BSD license and can be used for any purpose [6]. The library contains efficient implementations of both scalar and block Krylov-accelerated sparse solvers, with a range of different available preconditioners, including many variants of AMG, different smoothers (Jacobi, Gauss–Seidel), and partial factorizations. Quoting [6]: *AMGCL builds the AMG hierarchy on a CPU and then transfers it to one of the provided backends. This allows for transparent acceleration of the solution phase with help of OpenCL, CUDA, or OpenMP technologies. Users may provide their own backends which enables tight integration between AMGCL and the user code.*
- `AMGCL_CPRSolverAD`: Interface to the CPR implementation in AMGCL. The interface includes quasi-IMPES, true IMPES, and dynamic row-sum [8] types of reductions.
- `AMGCLSolverBlockAD` and `AMGCL_CPRSolverBlockAD` are variants of the regular AMGCL solvers requiring that the deferred-assembly option from Subsection 6.2.4 is enabled in the backend. These solvers eschew the use of regular sparse matrices and instead assemble directly into a custom block-CSR system that can be passed unmodified onto AMGCL. Because the linear systems are transferred directly without any postprocessing or copying, these are the fastest linear solvers available when applicable.
- There also exists a set of bindings to the DUNE iterative solver template library (DUNE-ISTL) [3]. At the time of writing, this is only experimental, but we hope to provide a fully functional interface in the not-too-distant future.

Performance Tests

The `linearSolversExample.m` script from `ad-core` demonstrates many of the solvers just described for a 3D test problem.

Problem setup: We again consider an $N \times N \times N$ Cartesian mesh, this time with four single-cell wells in the corners of the domain and a 1:10 vertical aspect ratio. We are only solving a linearized system, so the details of the setup are less important, but we note that the problem has significant density differences,

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

nonlinear relative permeabilities, and compressibility. We consider three different linear systems that may arise from simulating such a system: (i) the linearized fully implicit system, (ii) a pressure subproblem, and (iii) a transport subproblem. These can all be derived from the same underlying model equations as follows:

```
model  = GenericBlackOilModel(G, rock, fluid);
tmodel = TransportModel(model);
pmodel = PressureModel(model);
```

Here, the fully implicit model solves for pressure, water and gas saturation in each cell, and four variables per well; the pressure model only solves for the pressure in each cell; and the transport model solves for three saturations with a fixed total velocity. The fully implicit problem is of a mixed parabolic–hyperbolic type, whereas the pressure problem is purely parabolic and the transport is purely hyperbolic.

Solver setup: We let each solver use a maximum of 100 iterations to reach a strict tolerance of 10^{-6} in the residual norm.[3] We set up a few solvers for the fully implicit system: The MATLAB built-in `mldivide` solver; our MATLAB-based CPR solver with either `mldivide`, AGMG, or AMGCL as solvers for the pressure subproblem; and two AMGCL-CPR solvers. We use the default setup for most of the solvers and leave out the boilerplate setup code for brevity (full details are found in the script). We can examine the `AMGCL_CPRSolverAD` instance to see the default setup:

```
disp(cpr_cl)
```

```
AMGCL-CPR-block linear solver of class AMGCL_CPRSolverAD
-------------------------------------------------------
AMGCL constrained-pressure-residual (CPR) solver. Configuration:
           solver: bicgstab (Biconjugate gradient stabilized method.)
   preconditioner: amg (Algebraic multigrid)
       relaxation: spai0 (Sparse approximate inverse of order 0)
       coarsening: aggregation (Aggregation with constant interpolation)
                                  - aggr_eps_strong = 0.08
                                  - aggr_over_interp = 1
                                  - aggr_relax = 0.666667
     s_relaxation: ilu0 (Incomplete LU-factorization with zero fill-in - ILU(0))
                                  - ilu_damping = 1

->  AMGCL_CPRSolverAD with properties:
          doApplyScalingCPR: 1
          :
```

[3] In practice, simulators typically use less strict tolerances, because the nonlinear system requires many linearized systems to fully converge. Here, however, we employ a strict tolerance because some linear solvers have decreased rate of convergence after a few iterations because they only remove high-frequency errors, which may be misleading in terms of their general efficacy.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

The solver uses BiCGStab with aggregation AMG as preconditioner. The AMG preconditioner uses a sparse approximate inverse of the lowest order (SPAI0) as smoother, and because this is a CPR solver, there is a configuration present for a second-stage relaxation, which uses ILU(0) on the full system. In addition, there are a few hints of additional parameters specific to ILU(0) and aggregation coarsening that can be adjusted in `solver.amgcl_setup` or by variable input arguments to the constructor. If we would like to examine the possible options for, e.g., the coarsening, we can call the corresponding `set` routine without input arguments:

```
cpr_cl.setCoarsening()
```

```
No coarsening argument given. Available options:
smoothed_aggregation: Smoothed aggregation
ruge_stuben: Ruge-Stuben / classic AMG coarsening
aggregation: Aggregation with constant interpolation
smoothed_aggr_emin: Smoothed aggregation (energy minimizing)
```

We set up two versions of this solver: One that uses the regular sparse matrix representation and one that uses the block-CSR representation. The two are identical from a mathematical point of view, but the block format makes memory access more efficient.

For the last AMGCL-CPR solver, we can use some intuition of the problem to modify the defaults. For instance, the uniform permeability field is amenable to the alternative coarsening strategy of smoothed aggregation. It also seems likely that resolving the pressure to a strict tolerance is the most difficult part of the system, so we switch to multiple pre- and postsmoothing steps, with two cycles per level instead of the default of one. We can also explicitly set the solver to use the biconjugate gradient stabilized (BiCGStab) method as our outer solver:

```
cpr_mod = AMGCL_CPRSolverBlockAD(base_arg{:}, ...
             'aggr_eps_strong', 0.1, 'aggr_over_interp', 1.5, ...
             'npre', 1, 'npost', 2, 'ncycle', 2, 'id', '-bcsr-tweaked');
cpr_mod.setCoarsening('smoothed_aggregation')
cpr_mod.setRelaxation('spai0')
cpr_mod.setSolver('bicgstab')
```

Block solvers: The fully implicit system includes an additional preparation step for many of the solvers, which amounts to eliminating well equations via a Schur complement. The AMGCL-CPR solvers also require both a transpose and a reordering of the linear system to cell-major from the default variable-major ordering unless a block-CSR variant is used. For a system with two equations $\mathbf{R}_w, \mathbf{R}_o$ and two primary variables $\mathbf{p}, \mathbf{s}$ in each cell, we have

$$\mathbf{x}_{var} = [p_1, p_2, \ldots, p_n, s_1, s_2, \ldots, s_n] \quad \rightarrow \quad \mathbf{x}_{cell} = [p_1, s_1, p_2, s_2, \ldots, p_n, s_n].$$

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Table 6.4 *Linear solver time in seconds for a three-phase fully implicit problem.*

		8 000 cells		125 000 cells		421 875 cells		1 000 000 cells	
Solver	Req.	Total	Setup	Total	Setup	Total	Setup	Total	Setup
LU	–	2.49	0.02	576.58	0.18		–		–
CPR*	–	0.90	0.03	137.30	0.38		–		–
CPR*	AGMG	0.18	0.03	3.60	0.35	13.78	1.17	43.39	2.96
CPR*	AMGCL	0.21	0.03	3.44	0.36	16.20	1.18	51.35	3.24
CPR	AMGCL	0.07	0.02	0.43	0.02	3.38	1.11	10.20	3.12
CPR	AMGCL†	0.05	0.00	0.86	0.35	1.97	0.03	5.60	0.09
CPR	AMGCL‡	0.05	0.00	0.38	0.01	1.33	0.03	2.51	0.06

Here, $\star$ indicates that we are using `CPRSolverAD` (MATLAB) with another solver for the elliptic subsystem. Solvers marked with † and ‡ are both block-CSR solvers, with ‡ having algorithmic tweaks to improve performance for this test case.

The same reordering is also applied to the equations themselves, enabling the global Jacobian to be interpreted as a *block* Jacobian:

$$(J)^b_{ij} = \begin{bmatrix} \frac{\partial R_{wi}}{\partial p_j} & \frac{\partial R_{wi}}{\partial s_j} \\ \frac{\partial R_{oi}}{\partial p_j} & \frac{\partial R_{oi}}{\partial s_j} \end{bmatrix}. \tag{6.11}$$

This system is then applicable to block solvers; i.e., conventional scalar solvers converted by redefining elementary arithmetic operations, e.g., by replacing division by a number with a small block-sized matrix inverse. The default configuration of the AMGCL-CPR solver in MRST reflects the mixed nature of the system and treats the pressure system as scalar, with a block preconditioner for the whole system.

Performance comparison: The results are reported in Table 6.4 for 8 000 to 1 million cells, corresponding to 24 000 to 3 million cell-wise and 12 well degrees of freedom. We see that `mldivide` is by far the slowest for all but the smallest problem, using almost 500 seconds for the second smallest case; the fastest CPR solver uses only 1.3 seconds for this case. We generally see improvements when introducing a compiled elliptic solver for the MATLAB CPR solver and even more improvements as we go to the fully compiled AMGCL-CPR solvers, to the extent where it is difficult to recommend `CPRSolverAD` if a compiler is available. Switching to the block-CSR matrix significantly reduces runtime. We also note that the adjusted parameters for our tweaked AMGCL-CPR‡ solver results in a 50% improvement in solve time for the 1 million cells case.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Table 6.5 *Linear solver time in seconds for a three-phase pressure problem. The AMGCL variants all use variations of AMG to solve the system, whereas AGMG uses agglomeration only.*

	8 000 cells	125 000 cells	421 875 cells	1 000 000 cells
Solver	Total	Total	Total	Total
LU	0.06	0.27	32.6	–
AMGCL (classical)	0.03	0.18	0.83	1.80
AMGCL (aggregation)	0.04	0.21	0.78	2.31
AMGCL (smoothed aggregation)	0.05	0.17	0.55	1.47
AMGCL (energy minimization)	0.06	0.72	2.28	4.89
AGMG	0.03	0.23	0.75	1.77

The reported timings include overhead for the Schur complement and possible reordering for the block solvers that do not used deferred assembly. This is an area of possible future improvement, because the current implementation is written in pure MATLAB.

The solve times are generally much lower for the scalar pressure system reported in Table 6.5. Even a 1 million cell problem can be solved in just over a second with the optimal choice of AMG coarsening. The `mldivide` solver still has limited performance beyond the 10 000 cell range and cannot complete the largest model due to memory constraints. AGMG is algorithmically similar to AMGCL with aggregation and gives comparable performance. Although the transport subproblem, whose runtimes are reported in Table 6.6, has the same size as the fully implicit system, it is readily solved by all of the iterative solvers, even by the GMRES-ILU(0) solver implemented directly in MATLAB, which was not able to solve any of the fully implicit systems. Switching from a scalar ILU(0) with partial factorization to a block version significantly reduces the solve time.

Summary and recommendations: MRST has many general linear solvers available, but there can often be significant gains in adjusting your choice to the problem at hand. Systems that contain pressure (sub)systems are generally harder to solve than pure transport problems. Many solvers can be used in a black-box fashion and reasonable default choices for a given model can

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Table 6.6 *Linear solver time in seconds for a three-phase transport problem.*

	8 000 cells	125 000 cells	421 875 cells	1 000 000 cells
Solver	Total	Total	Total	Total
mldivide	0.50	127.4	–	–
mldivide	0.50	127.4	–	–
MATLAB-GMRES-ILU(0)	0.03	0.40	1.71	4.92
AMGCL-ILU(0)	0.03	0.44	1.48	3.57
AMGCL-block-GS	0.02	0.30	0.97	2.49
AMGCL-block-ILU(0)	0.02	0.27	0.91	2.32
AMGCL-block-ILU(0)†	0.01	0.09	0.34	0.90

The block solver marked with † uses the block-CSR representation of the system matrix.

be selected by `selectLinearSolverAD`, which is automatically called by `getNonLinearSolver` and `initEclipseProblemAD`, but spending some time testing different parameters can pay off for longer simulations. If your system exhibits a block structure, it is highly likely that even just a one-level block solver like GMRES-ILU(0) will significantly speed up your simulation. If you are working with large problems with high permeability contrasts, going to a multilevel method like AMG is highly recommended. There is always some degree of trade-off when performing an expensive setup phase that reduces the number of iterations; modern AMG variants like aggregation AMG provide a good balance between rigorous AMG hierarchy setup and computational performance if ILU(0) is not sufficient.

Installing and configuring a C++ compiler is worth the time for problems with more than a few thousand cells, especially if you wish to perform many simulations.

6.4 Setting Up and Managing Simulation Cases

Most examples and tutorials included with MRST are by design fairly small so that they can be run quickly to demonstrate functionality. From a conceptual point of view, there is no difference between setting up a model with 10 cells and a model with 1 million cells. In practice, however, larger simulation cases have additional requirements. Earlier in this chapter, you saw that models with a large number of degrees of freedom can be efficiently assembled and solved in MRST. In this section, we discuss how to efficiently *manage* single or multiple cases, with automatic restarts of aborted simulations and storage and retrieval of simulation results. We also explain how you can quickly set up a simulation from an ECLIPSE input file

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

and use functionality from MRST to choose intelligent defaults for the configuration of nonlinear and linear solvers.

6.4.1 Packed Problems: Storing and Running Simulation Cases

Many tutorials in MRST and several examples in the MRST textbook [14] use the main simulator interface `simulateScheduleAD`. It is efficient at simulating an entire *schedule* of time steps but will by default only return results once the simulation is complete. If the simulation stops due to a convergence failure at step 99 out of a 100, the first 99 steps are lost. The `ad-core` module supplies the `ResultHandler` class to automatically store intermediate results to disk[4] and it is possible to trigger the restart of a simulation from intermediate results through the `restartStep` optional argument. As an alternative to fine-grained manual calls to the simulator, we have introduced a concept we refer to as *packed problems* that separates the configuration and setup of a case from the simulation itself. We will demonstrate the basic functionality in this section before we use the functionality extensively in the examples. To this end, we use a pair of simple simulations.

Problem specification: We use a comparable setup to example 12.1.1 from [14], which in turn corresponds to `adBuckleyLeverett1D.m` from `ad-core`. This example can be found in `ad-core` as `demoPackedProblems.m`. We have a one-dimensional domain, 1 000 m long and discretized into 100 cells. We inject one pore volume of a fluid over 10 years, displacing the resident fluid that initially fills the domain. The fluid model is incompressible and immiscible, with equal viscosities for both phases. Let us say that we would like to see how this scenario behaves with two different relative permeability models:

```
fluid_1 = initSimpleADIFluid(..'n', [1, 1]..); % Linear relperm
model_1 = TwoPhaseOilWaterModel(G, rock, fluid_1);
fluid_2 = initSimpleADIFluid(..'n', [2, 2]..); % Quadratic relperm
model_2 = TwoPhaseOilWaterModel(G, rock, fluid_2);
```

We omit setup of initial state and schedule, which is immaterial to the discussion.

Setting up a packed problem: To use the packed problems, we must have a unique identifier for the case we are working with. Behind the scenes, all problems with the same name are stored in the same subfolder set up by the `mrstOutputDirectory()` utility. Keep in mind that your operating system must

[4] See example 12.1.1 in the MRST book [14] for details on using the `ResultHandler` class directly with `simulateScheduleAD`.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

be able to create and access a folder with this name and that there is sufficient space to store your results. We opt for a simple name in this case:

```
BaseName = 'test_packed';
```

Once we have the initial state, model, schedule, and a name for the scenario, we are ready to pack the pair of problems. The required input arguments of `packSimulationProblem` are the same as for `simulateScheduleAD`, followed by the case name and possibly a (short) description:

```
problem_1 = packSimulationProblem(state0, model_1, schedule, BaseName, ...
              'Name', 'linear_relperm', ...
              'Description', '1D displacement with linear flux');
problem_2 = packSimulationProblem(state0, model_2, schedule, BaseName, ...
              'Name', 'quadratic_relperm', ...
              'Description', '1D displacement with nonlinear flux');
```

The case name and the short description are optional, but the name is required here, because it will default to `class(model)`, which in our case would be the same for the two solvers. Additional optional arguments include `NonLinearSolver` and `ExtraArguments`, which are passed onto `simulateScheduleAD` when simulating. We can examine a packed problem to see the structure:

```
disp(problem_1)
```

```
          BaseName: 'test_packed'
              Name: 'linear_relperm'
       Description: '1D displacement with linear flux'
    SimulatorSetup: [1×1 struct]
           Modules: 1×4 cell
    OutputHandlers: [1×1 struct]
```

Apart from the already discussed fields, the `SimulatorSetup` field contains the inputs to `simulateScheduleAD`, the `Modules` field contains the list of loaded modules at initialization, and `OutputHandlers` contains `ResultHandlers` for all outputs. A packed problem then represents the entire definition of a simulation, including initial conditions, timestepping, and solvers. As we can see, it is fairly simple to convert a call to `simulateScheduleAD` to a packed problem.

Simulating packed problems: Now that the problems have been properly set up and packed, it is time to *solve* them:

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

```
problems = {problem_1, problem_2};
[ok, status] = simulatePackedProblem(problems);
```

Here, we have wrapped both problems in a cell array. We could alternatively have called `simulatePackedProblem` with a single problem as the input. The simulations are then performed sequentially, starting with the first problem:

```
**************************************************
* Case "test_packed" (linear_relperm)            *
* Description: "1D displacement with linear flux"  *
**************************************************
-> No output found, starting from first step...
Solving timestep 001/108: -> 3 Hours, 1518 Seconds, 750.00 Milliseconds
Solving timestep 002/108 ...
```

If the simulation is aborted for any reason – for instance if you sent an interrupt or your laptop ran out of battery – you can rerun the example script up to the same point to get an automatic restart:

```
-> Partial output found, starting from step 3 of 108...
Solving timestep 003/108 ...
```

If the entire case was already simulated, this will be acknowledged as well:

```
-> Complete output found, nothing to do here.
```

We can retrieve the simulation results by either accessing the `ResultHandler` instances in the packed problems or using routines that act directly on problems. One approach gives identical ordered outputs as `simulateScheduleAD`:

```
[ws, states, reports] = getPackedSimulatorOutput(problem_1)
```

This routine will give you results even if the simulation was not complete, enabling visualization of partial simulation results. You can even work with the results of an ongoing simulation from a different session if the same problem is present in both MATLAB instances. If you would like to rerun the simulation, you can either specify the optional `'restartStep'` argument to `simulatePackedProblem` or remove the stored data prior to simulation via the file system or function calls:

```
simulatePackedProblem(problem_1, 'restartStep', 1)   % Restart from first step
clearPackedSimulatorOutput(problems, 'prompt', true/false)   % Remove results
```

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

If the prompt input is not set to false, the command window will query before deleting each case to make sure you really want to throw away the fruits of the simulator's labors:

```
Do you want to delete 108 states and reports
for test_packed [linear_relperm]? y/n [n]: y
Removing files... Files removed.
```

6.4.2 Automatic Setup of ECLIPSE DataSets

If your simulation case is not set up from within MRST, chances are that it came from an ECLIPSE input deck. Section 12 of the MRST textbook [14] describes a set of routines you can use to make a script that runs simulations based on an input deck. MRST provides a pair of convenience routines for setting up such cases with reasonable defaults without having to manually read and parse the deck and construct the model yourself:

```
[state0, model, schedule, nls] = initEclipseProblemAD(deck);
```

Here, `deck` can either be a struct that contains the parsed `DATA` file or simply a string containing the path and name of the `DATA` file. The routine will initialize the initial state struct, pick the appropriate model and a suitable AD backend, and set up an appropriate nonlinear solver with automatic timestep selection and the best guess at a working and fast linear solver by calling `getNonLinearSolver`. The routine has a sibling, which enables you to set up an entire case directly into a packed problem:

```
problem = initEclipsePackedProblemAD(deck);
```

By taking advantage of these routines, scripts for running ECLIPSE cases can be quite short. The following example sets up SPE 9, simulates it with reasonable acceleration, and plots both the well and reservoir results:

```
mrstModule add ad-blackoil ad-core mrst-gui ad-props deckformat
fn      = fullfile( getDatasetPath('spe9'), 'BENCH_SPE9.DATA');
problem = initEclipsePackedProblemAD(fn, 'useMex', true, 'rowMajorAD', true);
simulatePackedProblem(problem, 'restartStep', 1);   % Simulate!
plotPackedProblem(problem);                         % Plot
```

The code is somewhat conservative in its default choices, so here we have enabled the row-major option and enabled `useMex` to indicate that MRST can pick options

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

that require a C++ compiler to get better performance. The packed problem automatically retrieves the title from the input file:

```
disp(problem)
```

```
  struct with fields:
          BaseName: 'SPE 9TH COMPARATIVE STUDY'
              Name: 'GenericBlackOilModel'
       Description: 'GenericBlackOilModel'
    SimulatorSetup: [1×1 struct]
           Modules: 1×7 cell
    OutputHandlers: [1×1 struct]
```

The routine uses the default output location configured by the utility function `mrstOutputDirectory`. Hence, rerunning a problem made from the same deck will automatically restart any simulation in progress. We can examine the properties of the packed problem to see the choices made:

```
disp(problem.SimulatorSetup.NonLinearSolver.LinearSolver)
```

```
AMGCL-CPR-block linear solver of class AMGCL_CPRSolverAD
  :
```

```
disp(problem.SimulatorSetup.NonLinearSolver.timeStepSelector)
```

```
IterationCountTimeStepSelector with properties:
  :
```

Our development policy is to keep new functionality out of the automatic deck initialization until it is considered fully stable. For instance, the deferred assembly option is not yet exposed as an option. Similarly, the default nonlinear tolerances and timestep strategies are fairly conservative. Modifying the model and nonlinear solver after set up is often a good approach to fine-tune the simulator for a specific case, leaving other options set to reasonable defaults.

6.5 Numerical Examples

We will consider two numerical examples that highlight how to benefit from many of the features discussed in the earlier sections. The first example illustrates how to use the concept of packed problems to simulate an ensemble of 1D models. In the second example, we compare the computational efficiency for different AD backends for a highly resolved sector model with more than 1 million cells.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

6.5.1 Packed Problems: Simulation of an Ensemble

We have seen that packing simulation problems makes it easy to store and retrieve simulation results between MATLAB sessions. We will next show how you can use packed problems to work with an ensemble of many closely related models; e.g., for use with uncertainty quantification.

Ensemble model: Complete source code for this example is found as `ensemblePackedProblemsExample` in `ad-core`. We consider the same 1D domain as in the previous section, with quadratic relative permeability functions and a 5:1 viscosity ratio between the resident and displacing fluid. To include the effect of permeability on the flow, both fluids are set to be compressible. Next, we generate 50 different porosity fields using `gaussianField` to give values between 0.01 and 0.5. The permeability is then generated from the porosity using the Carman–Kozeny relation. Parameters for specific surface area and grain size are taken from the MRST textbook [14, subsection 2.5.2]. Both the permeability and porosity are stored in matrices, with dimensions `G.cells.num` $\times$ 50. We loop over the rows and create a simulation model for each realization of permeability and porosity:

```
for i = 1:n
    caseName = sprintf('Case %d', i);       % Simple name
    rock = makeRock(G, K(:, i), p(:, i));   % Get realization
    model = GenericBlackOilModel(G, rock, fluid, 'gas', false);
    description = sprintf('Average porosity %1.2f, average perm %1.2f md', ...
                      mean(p), mean(k)/(milli*darcy));
    problems{i} = packSimulationProblem(state0, model, schedule, baseName, ...
                      'Name', caseName, 'Description', description);
end
```

There is a base name (`baseName`) for the entire case that groups the cases together, with case names and descriptions assigned to each ensemble member. We note that we could equally well have changed the schedule and the initial state for each case if these vary from one realization to another. We assume the initial conditions and injected volumes to be the same for all realizations and omit this configuration, which is identical to the previous example.

Batch simulation: We now have a set of problems representing our entire ensemble, which we can simulate with `simulatePackedProblem`. There are, however, a large number of realizations, and running them sequentially may take a long time. Modern computers usually have a number of CPU cores, and whereas many parts of MRST are parallel, serial parts will limit the total speedup during simulation. We can instead run the simulations in separate threads:

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

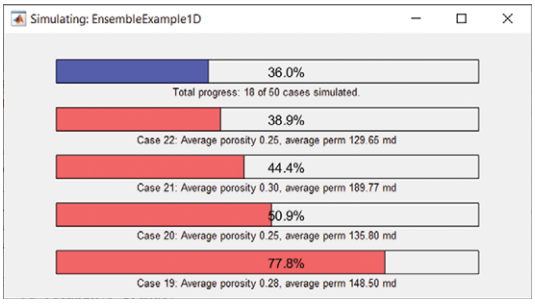

Figure 6.8 Visual progress monitoring for the ensemble case with simulations running in parallel as separate threads. The screenshot is taken on a PC with four available cores so that there are only 4 out of the 50 total cases running simultaneously.

```
ppm = PackedProblemManager(problems);     % Create manager class
ppm.simulateProblemsBatch();              % Run simulations in the background
```

After a few moments, you will be greeted by a sight similar to Figure 6.8. Four parallel simulations are running, corresponding to the default choice of one simulation per available core on this particular CPU. As each case finishes, a new simulation will be launched, ensuring that there are always four cases running if possible, until the entire ensemble has been simulated. This functionality is available with a basic MATLAB license, and MRST does not rely on the parallel computing toolbox to achieve this. Instead, a separate session without the graphical user interface is launched for each simulation. MRST stores each packed problem as a `.mat`-file together with details on which modules should be loaded. The main MATLAB session will launch additional sessions as needed and continue to update the progress bars. Each session loads MRST with `startup.m`, reads the packed problem, and performs a simulation before quitting. Hence, there is some startup cost to launching a background session, which for very small cases will outweigh the benefits.

Once one or more simulations have finished, we can extract results in bulk with a single call:

```
[ws, states, reports, names] = getMultiplePackedSimulatorOutputs(problems);
```

Instead of giving outputs for a single realization, this function gives us *all results* as cell arrays of cell arrays. We can thus find the state corresponding to timestep 10 of realization 25 in `states{25}{10}`. The function has several useful features for working with many similar simulation problems, including the output of

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

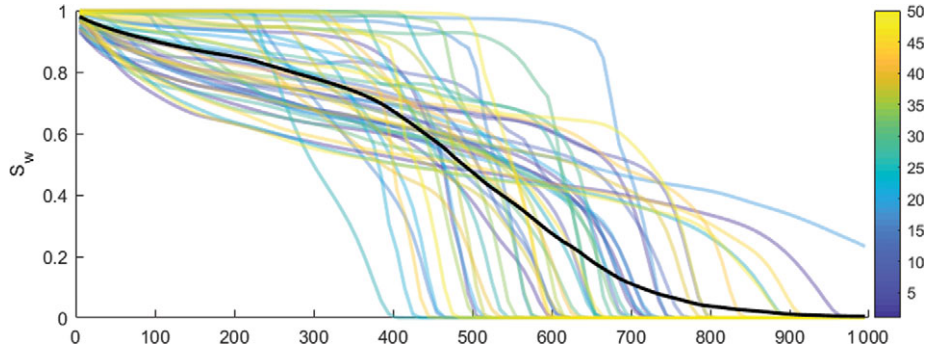

Figure 6.9 Water saturation for a specific timestep for the ensemble case: 50 different saturation distributions, each corresponding to one permeability and porosity realization are shown in different colors. The mean saturation over the entire ensemble is plotted as a black line.

different timesteps and grids corresponding to each subproblem, as well as returning `ResultHandler` instances to avoid reading all results into memory at once.

After we have retrieved the output from the entire ensemble, we can easily work with the data. We are not going to perform a detailed analysis of this synthetic example. Instead, we plot the solutions of all 50 ensembles for one timestep in Figure 6.9 together with the average water saturation for the entire ensemble. As expected, the large variation in petrophysical can have a large impact on the flow behavior for cases sharing the same fluid model.

6.5.2 Bringing It All Together: Running a Big Model

For our final example, we consider a larger variant of the model used for upscaling in [14, subsection 15.6.3]. In `blockAssemblyBigModelExample` we define a three-phase flow scenario on a $100 \times 100 \times 120$ grid with a total of 1 017 960 fine cells shown as in Figure 6.10. During simulation, the model has 3 053 880 degrees of freedom that describe the reservoir state. The fluid phases are compressible with nonlinear relative permeabilities, and the model is initialized at equilibrium by specifying water–oil and gas–oil contacts. We operate the three producers at fixed bottom-hole pressure and inject 0.25 pore volumes in a single injector over a 10-year period with 30-day timesteps. Just as when we tested the linear solvers, the exact details of the flow scenario are not of high importance.

The scenario is set up to use the `GenericBlackOilModel` with three different backends: sparse, row-major diagonal, and row-major diagonal with deferred assembly. All models use the same formulation for CPR: GMRES with aggregation AMG for the pressure subproblem, block ILU(0) for the global preconditioner, and quasi-IMPES pressure reduction with a tolerance of 10^{-3}. Aside

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

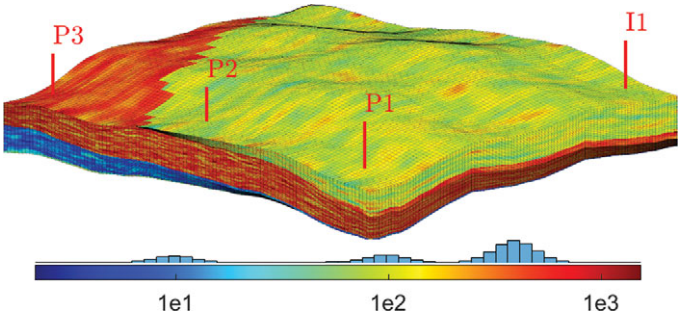

Figure 6.10 A larger model with 1 017 960 cells used in Example 6.5.2. The original version of this model with fewer grid cells is found in subsection 15.6.3 of the MRST textbook [14].

from the tolerance being loosened from 10^{-4}, this is the default setup in MRST. The diagonal block version with deferred assembly replaces the standard scalar solver with the `AMGCL_CPRSolverBlockAD` version. It is likely that further fine-tuning of the linear solver for this case could result in better results than the defaults.

When setting up multiple problems that share many parameters, it is often useful to define an anonymous helper function at the start of the script to ensure that all problems are otherwise identical:

```
packer = @(model, name, varargin) packSimulationProblem(state0, model, schedule,...
                                  'BigSectorExample', 'name', name, varargin{:});
```

Once set up, each of the models is packed as a problem and simulated with the sequential interface, making use of the full resources of our PC:

```
sparse_problem = packer(model_sparse, 'sparse-backend',   'NonLinearSolver', nls);
diag_problem   = packer(model_diag,   'diagonal-backend', 'NonLinearSolver', nls);
block_problem  = packer(model_bdiag,  'block-backend',    'NonLinearSolver', nls);
problems       = {sparse_problem, diag_problem, block_problem};
simulatePackedProblem(problems);
```

Each of the cases produces the exact same results, converging in the exact same number of nonlinear iterations but takes a different amount of time to do so. We retrieve the timing of each simulation:

```
timings = cellfun(@(x) getReportTimings(x, 'total', true), reports);
d = arrayfun(@(x) [x.Assembly./x.NumberOfAssemblies, ...
                   [x.LinearSolve, x.LinearSolvePrep]./x.Iterations],...
                   timings, 'UniformOutput', false);
```

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

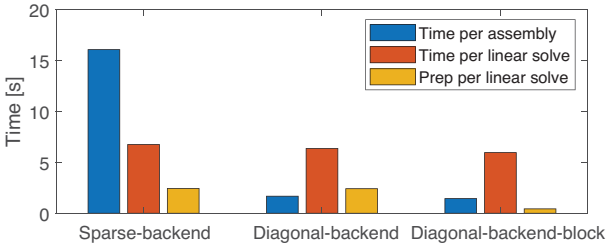

Figure 6.11 The time spent per assembly and per nonlinear iteration for a 1 million cell three-phase model with different AD versions.

Then, we plot the assembly time per linearization and the linear solver time for each nonlinear iteration:

```
bar(vertcat(d{:})) set(gca, 'XTickLabel', names)
legend('Time per assembly', 'Time per linear solve', 'Prep per linear solve')
ylabel('Time [s]')
```

Note that there are fewer iterations than assemblies, because the assembled discrete residual equations are not passed onto the linear solver if a timestep has converged. Figure 6.11 shows a graphical display of the timings. The same data are shown in Table 6.7, where we see that the sparse backend uses 17.9 seconds, the diagonal backend 1.8 seconds, and the block version 1.3 seconds per assembly with four threads. For comparison, using the C++ OPM Flow simulator [21] to simulate the same model consumes 1.1 seconds per assembly with four threads on the same CPU. The major benefit of the block version is that the preparation step for the linear solver avoids concatenating, reordering, and transposing the linear system before it is passed onto AMGCL, thereby reducing the time spent from 2.4 to 0.6 seconds per solve. The solvers generally spend a comparable amount of time on the linear solve itself, which limits the overall speedup.

Switching from sparse to the diagonal backend gives us a total speedup of 2.8, and using the block-diagonal version brings this up to a factor 4.3. If we examine the speedup of the assembly only, switching to diagonal variants nets us approximately one order of magnitude speedup. By switching the backend to the fastest available, we can reduce the total simulation time from 9 hours and 15 minutes to just under 3 hours for the whole case. The speedup would be much larger if we started with a less efficient linear solver, but using, e.g., a direct solver for this case would be somewhat disingenuous, because it would easily take a week to simulate. We also remark that the new *generic* models from Chapter 5 are in general faster than the original family of black-oil models discussed in the MRST textbook [14], because the former use vectorized code for wells and

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Table 6.7 *Breakdown of runtime in seconds for a three-phase model on a grid with 1 million cells simulated with sparse, diagonal, and block-diagonal assembly. All solvers use exactly the same number of nonlinear iterations. Speedup is reported relative to the default sparse backend.*

Backend		Time (total)	Time (each)	Speedup	% of total
Sparse (baseline)	Assembly	33 362.3	17.9	1.0	72.79
	Preparation	3 805.3	2.4	1.0	8.30
	Linear solve	7 493.5	4.7	1.0	16.35
	Total	*45 832.9*	*28.5*	*1.0*	*100.00*
Diagonal	Assembly	3 419.3	1.8	9.8	20.89
	Preparation	3 930.7	2.4	1.0	24.02
	Linear solve	7 690.3	4.8	1.0	46.99
	Total	*16 366.7*	*10.2*	*2.8*	*100.00*
DiagonalBlock	Assembly	2 455.5	1.3	13.6	22.97
	Preparation	911.2	0.6	4.2	8.52
	Linear solve	6 535.4	4.1	1.1	61.14
	Total	*10 690.0*	*6.6*	*4.3*	*100.00*

state functions that avoid redundant recomputations during assembly. In the end, assembly and related routines make up approximately 20% of the total runtime, which makes further improvements subject to diminishing returns. Speeding up this simulation significantly from this point would likely involve a combination of adjusting the timesteps, tweaking the linear solver settings, and changing the numerical method in use.

6.6 Concluding Remarks

We have seen that there are several ways to accelerate simulations in MRST. Optimized AD backends and improved linear solvers significantly reduce the time spent simulating. Packed simulation problems automatically store intermediate simulation results and can restart aborted simulations automatically so that you can restart your MATLAB session or tweak the solver parameters during a long simulation. Finally, whereas MRST is not intended as a platform for high-performance computing, the ongoing efforts demonstrated herein to improve performance make it possible to use the framework for problems with more than 1 million degrees of freedom and get a level of performance that is quite surprising when balanced against the flexibility MRST provides.

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

Appendix A Compilation of MRST Extensions

A large part of this chapter concerns improved execution speed. For some of the AD techniques described in Section 6.2, a C++ compiler must be available to MATLAB. The same applies for the AMGCL linear solvers from Section 6.3. A major advantage of using an integrated development environment such as MATLAB for writing numerical code is to avoid the myriad of compilation issues that are associated with external libraries and different compilers. Unfortunately, whereas MATLAB can be very efficient when using the high-level vectorized syntax correctly, some functions are not amenable to vectorization and compiled code may be needed for optimal performance. However, the thought of dealing with dependencies, platform-specific quirks, and build systems may intimidate potential users, and for this reason we have tried to streamline the process as much as possible:

- External dependencies are automatically downloaded from within MRST. For the linear solvers, this includes the AMGCL sources and the small required subset of the Boost library known to work with your specific release.
- Compilation should be supported by the free, license-permissive compilation option for a given MATLAB version. Specifically, this means GCC under GNU/Linux, the MinGW compiler on Windows, and Clang under OS X.
- The compilation should be performed directly from within MATLAB as needed.

As a user, you will nonetheless need to make sure that MATLAB has a working C++ compiler available. For details on the available options for your platform, please see the MathWorks help page on compilers [15]. To verify that you have a working C++ compiler available for MATLAB, please run `mex -setup C++` and follow the provided instructions.

All features described earlier in this chapter are automatically compiled as needed, but if you would like to get the compilation out of the way in a fresh install of MRST, you can manually trigger a build.

AD backends: For the AD operators described in Subsection 6.2.4:

```
mrstModule add ad-core
buildMexOperators();     % Build all operators that are not already compiled
buildMexOperators(true); % Force rebuild of all operators
```

It is also possible to build a specific named operator. To illustrate, we only build a single operator to reduce the amount of output:

```
buildMexOperators('names', 'mexDiscreteDivergenceJac');
```

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

```
Building MEX file mexDiscreteDivergenceJac...
Building with 'Microsoft Visual C++ 2019'.
MEX completed successfully.
Extension built in 4.87s -> OK!
```

Calling the function again with the same inputs results in no additional compilation, even if MATLAB is restarted:

```
buildMexOperators('names', 'mexDiscreteDivergenceJac');
```

```
mexDiscreteDivergenceJac is already compiled -> OK!
```

Linear solvers: The AMGCL solvers from Section 6.3 are compiled similarly:

```
mrstModule add linearsolvers
buildLinearSolvers();
```

```
Building with 'Microsoft Visual C++ 2019'.
AMGCL is compiled and ready for use.
```

MRST may in this process ask for permission to download the required source files, because these are not included with the released version of MRST. Note that the AMGCL interface in MRST includes a large number of different solvers and may take a few minutes to build. For this author, building linear solvers as just explained took slightly more than 2 minutes. The MEX extensions are also tentatively supported in GNU Octave, but there is some additional performance overhead as the MEX compatibility layer performs a copy of variables passed to the compiled executable. Extending MRST with the option to use the native Oct-file interface would remove this overhead, but this is not something we have been able to prioritize at present.

Appendix B Output from AD Benchmark

The output for the sparse reference and the diagonal row-major MEX backends in Subsection 6.2.5 for the 1 million cells model is shown here, demonstrating the backend performance on a large number of different tests together with significant speedup:

```
Backend #1 (baseline):
Sparse:
---------------------------
           Name | Time (s)
---------------------------
        cell_xy | 0.204016
        cell_xv | 0.099936
```

```
Backend #3:
Diagonal-MEX-RowMajor:
-------------------------------------
           Name | Time (s) | Speedup
-------------------------------------
        cell_xy | 0.012575 | 16.22
        cell_xv | 0.010842 | 9.22
```

https://doi.org/10.1017/9781009019781.011 Published online by Cambridge University Press

cell_xy_2z	0.268010
diagmult	0.101080
diagproductmult	0.245032
interp1_10	0.122377
interp1_50	0.124154
interp1_1000	0.127021
sparse	0.000389
subset_small	0.037223
subasgn_small	0.123621
subset_med	0.039283
subasgn_med	0.124855
subset_large	0.054378
subasgn_large	0.157129
faceavg	0.233785
Grad	0.247267
upw	0.292929
face_xy	0.834371
face_xv	0.321489
face_xy_2z	1.164837
Div	0.508060
AccDiv	0.671153
Total time	6.102395

cell_xy_2z	0.031201	8.59
diagmult	0.007379	13.70
diagproductmult	0.009892	24.77
interp1_10	0.014856	8.24
interp1_50	0.017938	6.92
interp1_1000	0.024308	5.23
sparse	0.025759	0.02
subset_small	0.000087	429.88
subasgn_small	0.012634	9.78
subset_med	0.000499	78.77
subasgn_med	0.013550	9.21
subset_large	0.007671	7.09
subasgn_large	0.017950	8.75
faceavg	0.042958	5.44
Grad	0.048814	5.07
upw	0.042245	6.93
face_xy	0.051084	16.33
face_xv	0.043241	7.43
face_xy_2z	0.143504	8.12
Div	0.160009	3.18
AccDiv	0.161755	4.15
Total time	0.900749	6.77

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, pp. 483–485, Academic Press, London, UK, 1967. doi: 10.1145/1465482.1465560.

[2] K. Bao, K.-A. Lie, O. Møyner, and M. Liu. Fully implicit simulation of polymer flooding with MRST. *Computational Geosciences*, 21(5–6):1219–1244, 2017. doi: 10.1007/s10596-017-9624-5.

[3] M. Blatt and P. Bastian. The iterative solver template library. In B. Kagström et al., eds., *International Workshop on Applied Parallel Computing*, pp. 666–675. Springer, Berlin, 2006. doi: 10.1007/978-3-540-75755-9_82.

[4] H. Cao, H. A. Tchelepi, J. R. Wallis, and H. E. Yardumian. Parallel scalable unstructured CPR-type linear solver for reservoir simulation. In *SPE Annual Technical Conference and Exhibition, Dallas, TX, 9–12 October*. Society of Petroleum Engineers, 2005. doi: 10.2118/96809-MS.

[5] D. DeBaun et al. An extensible architecture for next generation scalable parallel reservoir simulation. In *SPE Reservoir Simulation Symposium*, 2005. doi: 10.2118/93274-MS.

[6] D. Demidov. C++ library for solving large sparse linear systems with algebraic multigrid method. URL https://github.com/ddemidov/amgcl.

[7] D. Demidov. AMGCL: an efficient, flexible, and extensible algebraic multigrid implementation. *Lobachevskii Journal of Mathematics*, 40(5):535–546, 2019. doi: 10.1134/S1995080219050056.

[8] S. Gries, K. Stüben, G. L. Brown, D. Chen, and D. A. Collins. Preconditioning for efficiently applying algebraic multigrid in fully implicit reservoir simulations. *SPE Journal*, 19(4):726–736, 2014. doi: 10.2118/163608-PA.

[9] J. D. Jansen. Adjoint-based optimization of multi-phase flow through porous media—a review. *Computers & Fluids*, 46(1):40–51, 2011. doi: 10.1016/j.compfluid.2010.09.039.

[10] J. Killough. Ninth SPE comparative solution project: a reexamination of black-oil simulation. In *SPE Reservoir Simulation Symposium, 12–15 February, San Antonio, Texas*. Society of Petroleum Engineers, 1995. doi: 10.2118/29110-MS.

[11] J. Killough and C. Kossack. Fifth comparative solution project: evaluation of miscible flood simulators. In *SPE Symposium on Reservoir Simulation, 1–4 February, San Antonio, Texas*. Society of Petroleum Engineers, 1987. doi: 10.2118/16000-MS.

[12] S. Krogstad, K.-A. Lie, O. Møyner, H. M. Nilsen, X. Raynaud, and B. Skaflestad. MRST-AD – an open-source framework for rapid prototyping and evaluation of reservoir simulation problems. In *SPE Reservoir Simulation Symposium, 23–25 February, Houston, Texas, USA*. Society of Petroleum Engineers, 2015. doi: 10.2118/173317-MS.

[13] X. Li and D. Zhang. A backward automatic differentiation framework for reservoir simulation. *Computational Geosciences*, 18(6):1009–1022, 2014. doi: 10.1007/s10596-014-9441-z.

[14] 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.

[15] MathWorks. Compilers – MATLAB & Simulink. URL https://mathworks.com/support/requirements/supported-compilers.html.

[16] MathWorks. IR array of sparse array – MATLAB. URL https://se.mathworks.com/help/matlab/apiref/mxsetir.html.

[17] O. Møyner. Next generation multiscale methods for reservoir simulation. PhD thesis, 2016. URL http://hdl.handle.net/11250/2431831.

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

[19] Y. Notay. AGMG software and documentation. URL https://agmg.eu/.

[20] Y. Notay. An aggregation-based algebraic multigrid method. *Electronic Transactions on Numerical Analysis*, 37(6):123–146, 2010.

[21] A. F. Rasmussen, T. H. Sandve, K. Bao, A. Lauser, J. Hove, B. Skaflestad, R. Klöfkorn, M. Blatt, A. B. Rustad, O. Sævareid, K.-A. Lie, and A. Thune. The open porous media flow reservoir simulator. *Computers & Mathematics with Applications*, 81:159–185, 2021. doi: 10.1016/j.camwa.2020.05.014.

[22] D. V. Voskov and H. A. Tchelepi. Comparison of nonlinear formulations for two-phase multi-component EoS based simulation. *Journal of Petroleum Science and Engineering*, 82–83:101–111, 2012. doi: 10.1016/j.petrol.2011.10.012.

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

[24] J. R. Wallis. Incomplete Gaussian elimination as a preconditioning for generalized conjugate gradient acceleration. In *SPE Reservoir Simulation Symposium, 15–18 November, San Francisco, California*. Society of Petroleum Engineers, 1983. doi: 10.2118/12265-MS.

[25] R. Younis. Advances in modern computational methods for nonlinear problems; a generic efficient automatic differentiation framework, and nonlinear solvers that converge all the time. PhD thesis, Stanford University, Stanford, CA, 2009.

[26] R. Younis and K. Aziz. Parallel automatically differentiable data-types for next-generation simulator development. In *SPE Reservoir Simulation Symposium,*

26–28 February, Houston, Texas, U.S.A., Society of Petroleum Engineers, 2007. doi: 10.2118/106493-MS.

[27] Y. Zhou, H. A. Tchelepi, and B. T. Mallison. Automatic differentiation framework for compositional simulation on unstructured grids with multi-point discretization schemes. In *SPE Reservoir Simulation Symposium, 21–23 February, The Woodlands, Texas, USA*, Society of Petroleum Engineers, 2011. doi: 10.2118/141592-MS.