Appendix

The MATLAB Reservoir Simulation Toolbox

Practical computer modeling of porous media constitutes an important part of the book and is presented through a series of examples that are intermingled with more traditional textbook material. All examples discussed in the book rely on the MATLAB Reservoir Simulation Toolbox (MRST), which is a free, open-source software that can be used for any purpose under the GNU General Public License version 3 (GPLv3).

The toolbox is primarily developed by my research group at SINTEF, which is the fourth-largest contract research organization in Europe. The software started out as a research code developed to study consistent discretization and multiscale solvers for incompressible two-phase flow on stratigraphic and unstructured polyhedral grids. Over the past 6–7 years the software has been applied to research a wide spectrum of other problems related to reservoir modeling, and, as a result, the software today has many of the same capabilities that can be found in commercial reservoir simulators. In addition, it contains a spectrum of new research ideas; some of these were discussed in Chapters 13 and 14. At least for the time being, our main motivation for continuing to maintain and develop the software is to have a versatile research platform that enables researchers at SINTEF to rapidly develop proof-of-concept implementations of new ideas and then subsequently, and rather effortlessly, turn these into software prototypes that could be verified and validated for problems of industry-standard complexity with regard to flow physics and geological description of geology. A second purpose is to support the idea of replicable/reproducible research, as well as to enable us to effectively leverage results from one research project in another.

Over the years, we have also seen that MRST is an efficient teaching tool and a good platform for disseminating new ideas. Maintaining and developing the software as a reliable community code takes a considerable effort, but by carefully documenting and releasing our research software as free, open source, we hope to contribute to simplifying the experimental programming of other researchers in the field and give a head start to students about to embark on a master's or PhD project.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

A.1 Getting Started with the Software

This Appendix provides you with a brief overview of the software and the philosophy underlying its design. I show you how to obtain and install the software and explain its terms of use, as well as how we recommend that you use the software as a companion to the textbook. I also briefly discuss how you can use a scripted, numerical programming environment like MATLAB (or its open-source clone GNU Octave) to increase the productivity of your experimental programming and share examples of tricks and ways of working with MATLAB that we have found particularly useful. I end the chapter by introducing you to automatic differentiation, which is one of the key aspects that make MRST a powerful tool for rapid prototyping and enable us to write compact and quite self-explanatory codes that are well suited for pedagogical purposes. As a complement to the material presented in this chapter, you should also consult the first of two JOLTS (short online learning modules), developed in collaboration with Stanford University [188]. The JOLT gives a brief overview of the software, shows the way it looked a few years ago, tells you why and how it was created, and instructs you how to download and install it on your computer. If you are not interested in programming at all, you need not read this Appendix. However, if you choose to not work with the software alongside the textbook material, be warned, you will miss a lot of valuable insight.

A.1.1 Core Functionality and Add-on Modules

MRST is a research tool whose aim is to support research on modeling and simulation of flow in porous media. The software contains a wide variety of mathematical models, computational methods, plotting tools, and utility routines that extend MATLAB in the direction of reservoir simulation. To make the software as flexible as possible, it is organized quite similar to MATLAB and consists of a collection of core routines and a set of add-on modules, as illustrated in Figure A.1. The material presented in Part I of the book relies almost entirely on general core functionality, which includes routines and data structures for creating and manipulating grids, petrophysical data, and global drive mechanisms such as gravity, boundary conditions, source terms, and wells. The core functionality also contains an implementation of automatic differentiation – you write the formulas and specify the independent variables, the software computes the corresponding derivatives or Jacobians – based on operator overloading (see Section A.5), as well as a few routines for plotting cell and face data defined over a grid. The core functionality is considered to be stable and not expected to change significantly in future releases.

To minimize maintenance costs and increase flexibility, MRST core does not contain flow equations, discretizations, and solvers; these are implemented in various add-on modules. If you have read Chapter 1, you have already encountered the `incompTPFA` solver from the `incomp` module in Section 1.4; this module implements fluid behavior and standard solvers for incompressible, immiscible, single-phase and two-phase flow. The mathematical models, discretizations, and solution techniques underlying this module

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

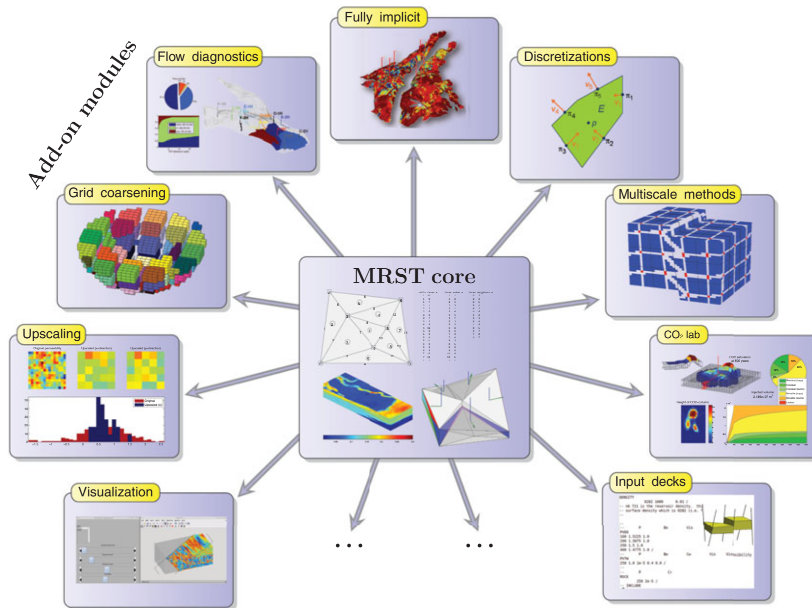

Figure A.1 MRST consists of core functionality that provides basic data structures and utility functions, and a set of add-on modules that offer discretizations and solvers, simulators for incompressible and compressible flow, and various workflow tools such as flow diagnostics, grid coarsening, upscaling, visualization of simulation output, and so on.

are extensively discussed in Parts II and III of the book. In particular, Chapters 5 and 10 outline key functionality offered in the module and discuss in detail how the solvers are implemented. The standard solvers are unfortunately not unconditionally consistent and may therefore exhibit strong grid-orientation errors and fail to converge. To get a convergent scheme, one can use one of the methods from the `mimetic` and `mpfa` modules discussed in Chapter 6, which offer consistent discretizations on general polyhedral grids [192].

Solvers for incompressible flow have been part of the software since the beginning and constitute the first family of add-on modules signified by the “Discretizations” block in Figure A.1. The `incomp`, `mimetic`, and `mpfa` modules are all implemented using a procedural (imperative) programming model from classical MATLAB, i.e., using mathematical functions that operate mainly on vectors, (sparse) matrices, structures, and a few cell arrays. These are generally robust and well documented, have remained stable over many years, and will not likely change significantly in future releases. The family of incompressible flow solvers also includes a few other discretization methods from the research front like virtual element methods and are intimately connected with two of the modules implementing multiscale methods (MsFV and MsMFE). These are not discussed herein; you can find details in the software itself, on the MRST website, or in the many papers using the software.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

The second family of modules consists of simulators based on automatic differentiation and is illustrated by the "Fully implicit" block in Figure A.1. The introduction of automatic differentiation has been a big success and has not only enabled efficient development of black-oil simulators, but also opened up for a unprecedented capabilities for rapid prototyping [170, 30]. The process of writing new simulators is hugely simplified by the fact that you no longer need to compute analytic expressions for derivatives and Jacobians. This is discussed in Chapters 7 and 11. Initially, we implemented solvers based on automatic differentiation using an imperative programming model similar to the `incomp` family of modules. Soon it became obvious that this was a limiting factor, and a new object-oriented programming model, implementing a general framework for solvers referred to as MRST AD-OO [170, 212, 30], was introduced. The individual modules and the underlying implementations of AD-OO underwent significant changes in the period 2014–2016. From release 2016a and onward, however, the basic functionality has remained largely unchanged and has mainly been subject to bug fixes, feature enhancements, and performance improvements.

The `ad-core` module is the most basic part in MRST AD-OO and does not contain any complete simulators, but rather implements the common framework used for many other modules. The design of this framework deviates significantly from that of the incompressible solvers. The main motivation for introducing an object-oriented framework is to be able to simulate compressible multiphase models of industry-standard complexity. Chapters 11 and 12 only discuss how to simulate compressible multiphase models of black-oil type, but the software also offers simulators for compositional flow. All these multiphase models are significantly more complex to simulate than the basic incompressible models implemented in the `incomp` module. Not only are there more equations and more complex parameters and constitutive relationships, but making robust simulators also necessitates more sophisticated solution algorithms involving nonlinear solvers, preconditioners, time-step control, variable switching, etc. Moreover, industry-standard simulations generally require lots of bells and whistles to implement specific fluid behavior, well models, and group controls. A robust implementation also requires a number of subtle tricks of the trade to ensure that your simulator is able to reproduce results of leading commercial simulators. Object-orientation enables us to divide the implementation into different numerical contexts (mathematical model, nonlinear solver, time-step control, linearization, linear solver, etc.), hide unnecessary details, and only expose the details that are necessary within each specific context.

The third family of modules consists of tools that can be used as part of the reservoir modeling workflow. In Part IV, we go through the three types of tools shown in Figure A.1. "Diagnostics" signifies a family of computational methods for determining volumetric connections in the reservoir, computing well-allocation factors, measuring dynamic heterogeneity, providing simplified recovery estimates, etc. Tools from the "Grid coarsening" and "Upscaling" modules can be used to develop reduced simulation models with fewer degrees of freedom and hence lower computational costs. MRST also offers several other

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

modules of the same type, e.g., history matching and production optimization, but these are outside the scope of the book.

The fourth family of modules consists of computational methods that have been developed to study a special problem. In Figure A.1, this is exemplified by the "CO_2 lab" module, which is a comprehensive collection of computational methods and modeling tools developed especially to study the injection and long-term migration of CO_2 in large aquifer systems. Other modules of the same type include solvers for geomechanics and various modeling frameworks and simulators for fractured media. These modules are all outside the scope of this book.

The fifth family consists of a variety of utility modules that offer graphical interfaces and advanced visualization, more comprehensive routines for reading and processing simulation models and other input data, C-acceleration of selected routines from the core module to avoid computational bottlenecks, etc. You will encounter functionality from several of these modules throughout the book, but the modules themselves will not be discussed in any detail. Last, but not least, there are also modules developed by researchers not employed by SINTEF.

A.1.2 Downloading and Installing

The main parts of MRST are hosted as a collection of software repositories on Bitbucket. Official releases are provided as self-contained archive files that can be downloaded from the website: www.mrst.no/

Assume now that you have downloaded the tarball of one of the recent releases; here, we use release 2016b as an example. Issuing the following command

```
untar mrst-2016b.tar.gz
```

in MATLAB creates a new folder `mrst-2016b` in your current working director that contains all parts of the software. Once all code has been extracted to some folder, which we henceforth refer to as the *MRST root* folder, you must navigate MATLAB there, either using the built-in file browser, or by using the `cd` command. Assuming that the files were extracted to the home folder, this would amount to the following on Linux/Mac OS:

```
cd /home/username/mrst-2016b/          % on Linux/Mac OS
cd C:\Users\username\mrst-2016b\       % on Windows
```

To activate the software, you must make sure that the MRST root folder is on MATLAB's search path. This is done by use of a startup script, which also scans your installation and determines which modules you have installed. When you are in the folder that contains the software, you the software is activated by the command:

```
startup;
```

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

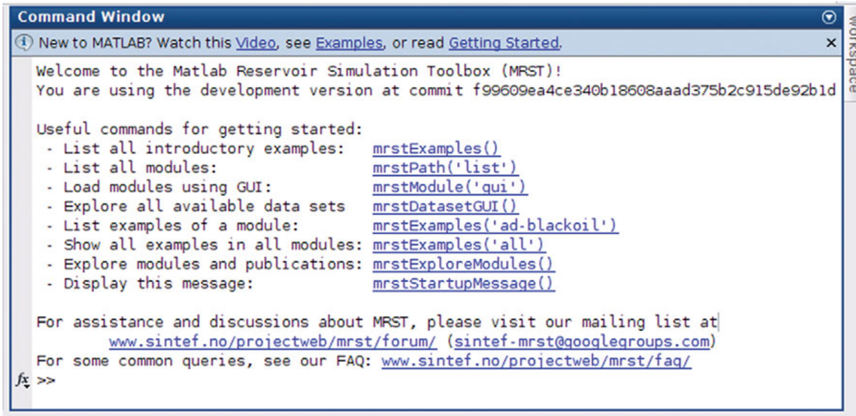

Figure A.2 The welcome message displayed when the `startup` script is run in release 2016a and later. The careful reader may notice that the user runs a development version of the software and not one of the official releases.

In MRST 2016a and newer, the startup script will display a welcome message showing that the software is initialized and ready to use; see Figure A.2. The whole procedure of downloading and installing the software, step by step, can be seen in the first MRST Jolt [188] (uses release 2014b).

At this point, a word of caution is probably in order. We generally refer to the software as a toolbox. By this we mean that it is a collection of data structures and routines that can be used alongside with MATLAB. It is, however, *not* a toolbox in the same sense as those purchased from the official vendors of MATLAB. This means, for instance, that MRST is not automatically loaded unless you start MATLAB from the MRST root folder, or make this folder your standing folder and manually issue the `startup` command. Alternatively, if you do not want to navigate to the root folder, for instance in an automated script, you can call `startup` directly

```
run /home/username/mrst-2016b/startup    % or C:\MyPath\mrst-2016b\startup
```

In versions prior to MRST 2016a, the startup script only sets up the global search path so that MATLAB is able to locate MRST's core functionality and the various modules. To verify that the software is working, you can run the simple example discussed in Section 1.4 by typing `flowSolverTutorial1`. This should produce the same plot as in Figure 1.4.

A.1.3 Exploring Functionality and Getting Help

The welcome message shown in Figure A.2 contains links to a number of functions that are useful if you want to get more acquainted with the software. Upon your first encounter,

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

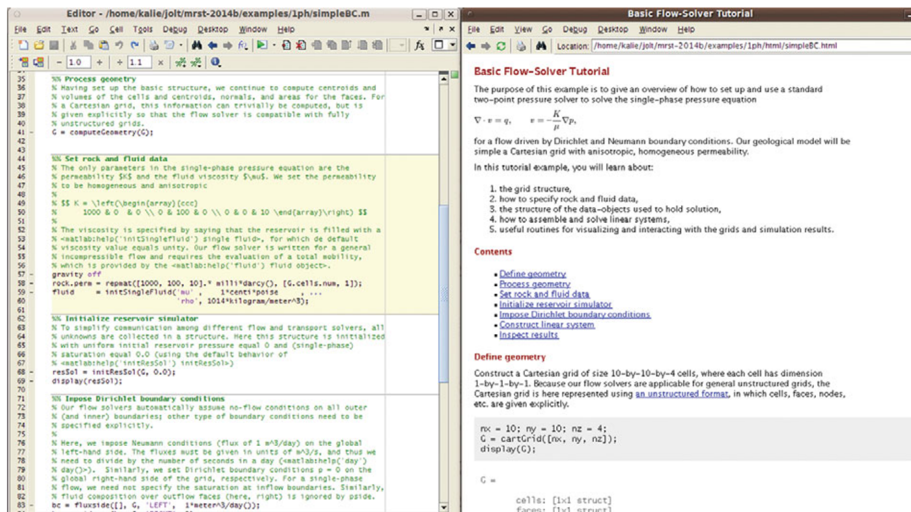

Figure A.3 Illustration of the MATLAB workbook concept. The editor window shows part of the source code for the `simpleBC` tutorial from release 2014b. (In newer versions, this tutorial has been renamed `incompIntro` and moved to the new `incomp` module.) Notice how cells are separated by horizontal lines and how each cell has a header and a text that describes what the cell does. The exception is the first cell, which summarizes the content of the whole tutorial. The right window shows the result of publishing the workbook as a webpage.

the best point to start is by typing (or clicking the corresponding blue text in the welcome message)

```
mrstExamples()
```

This will list all introductory examples found in MRST core. Some of these examples introduce you to basic functionality, whereas others highlight capabilities found in various add-on modules that implement specific discretizations, solvers, and workflow tools. These examples are designed using *cell-mode scripts*, which can be seen as a type of "MATLAB workbook" that enables you to break a script into smaller code sections (code cells) that can be run individually to perform a specific task such as creating parts of a model or making an illustrative plot; see Figure A.3 for an illustration.

In my opinion, the best way to understand tutorial examples is to go through the corresponding scripts, evaluating one section at the time. Alternatively, you can set a breakpoint on the first line and step through the script in debug mode, e.g., as shown in the fourth video of the first MRST Jolt [188]. Some of the example scripts contain quite a lot of text and are designed to be published as HTML documents, as shown to the right in Figure A.3. If you are not familiar with cell-mode scripts or debug mode, I strongly urge you to learn these useful features in MATLAB as soon as possible.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

```
>> help computeTrans
 Compute transmissibilities.

  SYNOPSIS:
    T = computeTrans(G, rock)
    T = computeTrans(G, rock, 'pn', pv, ...)

  PARAMETERS:
    G    - Grid structure as described by grid_structure.

    rock - Rock data structure with valid field 'perm'.  The permeability
           is assumed to be in measured in units of metres squared (m^2).
           Use function 'darcy' to convert from darcies to m^2, e.g.,
                  perm = convertFrom(perm, milli*darcy)
           if the permeability is provided in units of millidarcies.
           :
           :
  RETURNS:
    T - half-transmissibilities for each local face of each grid cell
        in the grid.  The number of half-transmissibilities equals
        the number of rows in G.cells.faces.

  COMMENTS:
    PLEASE NOTE: Face normals are assumed to have length equal to
    the corresponding face areas. ..

  SEE ALSO:
    computeGeometry, computeMimeticIP, darcy, permTensor.
```

Figure A.4 Most functions in MRST are documented in a standard format that gives a one-line summary of what the function does, specifies the synopsis (e.g., how the function should be called), explains input and output parameters, and points to related functions.

To find the syntax and understand what a specific function does, you can type

```
help computeTrans
```

which will bring up the documentation for `computeTrans` shown in Figure A.4. All core functionality is well documented in a format that follows the MATLAB standard. Functionality you are meant to utilize from any of the add-on modules should also, as a rule, be well documented. However, the format and quality of documentation may differ more, depending upon who developed the module and how mature and widely used it is. Starting with release 2017b, all documentation has been formatted so that it can be auto-extracted with Sphinx, e.g., as HTML posted on the MRST website (see Figure A.5).

As a general rule, all modules distributed as part of the MRST release are required to contain worked tutorials highlighting key functionality that most users should understand. A subset of these tutorials is also available on the MRST website. To list tutorial examples found in individual modules, you can use the `mrstExamples` command, which can also list all the tutorial examples the software offers,

```
mrstExamples('ad-blackoil')      % all examples in the black-oil module
mrstExamples('all')              % all examples across all available modules
```

To learn more about the different modules and get a full overview of all available functionality, you can type the command

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

The Matlab Reservoir Simulation Toolbox
2017
Search docs

Core functionality
Plotting routines
Grid generation, processing and manipulation
Parameters, states and forces
Input to solvers
Petrophysical properties
Wells, boundary conditions and source terms
Utilities
Units
Examples
MRST Release notes
ad-core: Automatic Differentiation Core
ad-props: PVT, fluid models and other properties for the AD-solvers
ad-blackoil: Black-oil solvers
ad-eor: Enhanced oil recovery solvers

Input to solvers

`computeTrans`(*G, rock, varargin*)

Compute transmissibilities.

Synopsis:

```
T = computeTrans(G, rock)
T = computeTrans(G, rock, 'pn', pv, ...)
```

Parameters:
- **G** – Grid structure as described by grid_structure.
- **rock** – Rock data structure with valid field `perm`. The permeability is assumed to be in measured in units of metres squared (m^2). Use function `darcy` to convert from darcies to m^2, e.g:

```
perm = convertFrom(perm, milli*darcy)
```

if the permeability is provided in units of millidarcies.

The field rock.perm may have ONE column for a scalar permeability in each cell, TWO/THREE columns for a diagonal permeability in each cell (in 2/3 D) and THREE/SIX columns for a symmetric full tensor permeability. In the latter case, each cell gets the permeability tensor:

```
K_i = [ k1  k2 ]      in two space dimensions
      [ k2  k3 ]

K_i = [ k1  k2  k3 ]  in three space dimensions
      [ k2  k4  k5 ]
      [ k3  k5  k6 ]
```

Figure A.5 Auto-generated documentation of all function, as well as all classes from the AD-OO framework, can also be found on the MRST website.

```
mrstExploreModules()
```

This brings up a graphical user interface that lists all accessible modules and outlines their purpose and functionality, including a short description of all tutorial examples found in each module, as well as a list of relevant scientific publications, with possibility to view online versions and export citations to BibTeX. Section A.3 gives a brief description of the majority of the modules.

MRST also offers simplified access to a number of public data sets. You can use the following graphical user interface to list all data sets that are known to MRST

```
mrstDatasetGUI()
```

The GUI briefly describes each data set, provides functionality for downloading and installing it in a standard location, lists the files it contains as well as the tutorial examples in which it is used. More details are given in Section A.2.

Last, but not least, a number of common queries have been listed on MRST's FAQ page: www.sintef.no/projectweb/mrst/faq/ For further assistance and discussions you may visit the user forum (www.sintef.no/projectweb/mrst/forum/) and/or subscribe to the mailing list (sintef-mrst@googlegroups.com). As you get more experience with the software, I encourage you to help others out by answering questions and contributing to discussions.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

A.1.4 Release Policy and Version Numbers

Over the last few years, key parts of the software have become relatively mature and well tested. This has enabled a stable biannual release policy with one release in the spring and one in the fall. The version number of MRST refers to the biannual release schedule and does not imply a direct compatibility with the same release number for MATLAB. That is, you do not need to use MRST 2016b if you are using MATLAB R2016b, or vice versa, you do not need to upgrade your MATLAB to R2019a to use MRST 2019a.

Throughout the releases, basic functionality like grid structures has remained largely unchanged, except for occasional and inevitable bug fixes, and our primary focus has been on expanding functionality by maturing and releasing in-house prototype modules. Fundamental changes will nevertheless occur from time to time, e.g., like when automatic differentiation was introduced in 2012 and when its successor, the AD-OO framework, was introduced in 2014b. Likewise, parts of the software may sometimes be reorganized, like when the basic incompressible solvers were taken out of the core functionality and put in a separate module in 2015a. In writing this, I (regretfully) acknowledge the fact that specific code details and examples in books describing evolving software tend to become somewhat outdated. To countermand this, complete codes for almost all examples presented in the book are contained in a separate `book` module that accompanies 2015a and later releases. These are part of the prerelease test suite and should thus always be up to date.

A.1.5 Software Requirements and Backward Compatibility

MRST was originally implemented so that the minimum requirement should be MATLAB version 7.4 (R2007a). However, certain parts of the software use features that were not present in R2007a:

- The AD libraries use new-style classes (`classdef`) introduced in R2008a.
- Various scripts use new syntax for random numbers introduced in R2007b.
- Use of the tilde operator to ignore return values (e.g., `[~,i]=max(X,1)`) was introduced in R2009b.
- Some routines, like the fully implicit simulators for black-oil models, rely on accessing sub-blocks of large sparse matrices. Although these routines will run on any version from R2007a and onward, they may not be efficient on versions older than R2011b.

When it comes to visualization, things are a bit more complicated, since MATLAB 3D graphics does not behave exactly the same on all platforms. Moreover, MATLAB introduced new handle graphics in R2014b, which has been criticized by many because it is slow and because it breaks backward compatibility. We have tried to revise plotting in newer versions of MRST so that it works well both with the old and the new handle

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

graphics, but every now and then you may stumble across certain tricks (e.g., setting grid lines semitransparent to make them thinner) that may not work well for your particular MATLAB version.

MRST can also be used to a certain extent with recent versions of GNU Octave, which is an open-source scientific programming language that is largely compatible with MATLAB. The procedural parts should work more or less out of the box, but you may encounter some problems with some of the (3D) plotting, which works a bit differently in GNU Octave. The 2017b release adds preliminary support also for the object-oriented AD-OO framework through the `octave` module. Unfortunately, the performance seems to be orders of magnitude behind recent versions of MATLAB, but this will hopefully improve in the future. The only parts of MRST you cannot expect to work are graphical user interfaces; their implementation is fundamentally different between MATLAB and GNU Octave.

MRST is designed to only use standard MATLAB. Nevertheless, we have found a few third-party packages and libraries to be quite useful:

- `MATLAB-BGL`: MATLAB does not yet have extensive support for graph algorithms. The MATLAB Boost Graph Library contains binaries for useful graph-traversal algorithms such as depth-first search, computation of connected components, etc. The library is freely available under the BSD License from the Mathwork File Exchange[1]. MRST has a particular module (see Section A.3) that downloads and installs this library.
- `METIS` is a widely used library for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices [154]. The library is released[2] under a permissive Apache License 2.0.
- `AGMG`: For large problems, the linear solvers available in MATLAB are not always sufficient, and it may be necessary to use an iterative algebraic multigrid method. AGMG [238] has MATLAB bindings included and was originally published as free open-source. The latest releases have unfortunately only offered free licenses for academic research and teaching[3].
- `AMGCL` is a header-only library for preconditioning with and without algebraic multigrid[4].

`MATLAB-BGL` is required by several of the more advanced solvers that are not part of the basic functionality in MRST. Installing the other packages is recommended but not required. When installing extra libraries or third-party toolboxes you want to integrate with MRST, you must make the software aware of them. To this end, you should add a new script called `startup_user.m` and use the built-in command `mrstPath` to make sure that the routines you want to use are on the search path used by MRST and MATLAB to find functions and scripts.

[1] http://www.mathworks.com/matlabcentral/fileexchange/10922
[2] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview
[3] http://homepages.ulb.ac.be/~ynotay/AGMG/
[4] http://github.com/ddemidov/amgcl

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

A.1.6 Terms of Usage

MRST is distributed as free, open-source software under the GNU Public License (GPLv3).[5] This license is a widely used example of a so-called copyleft license that offers the right to distribute copies and modified versions of copyrighted creative work, provided the same rights are preserved in modified and extended versions of the work. For MRST, this means that you can use the software for any purpose, share it with anybody, modify it to suit your needs, and share the changes you make. However, if you share any version of the software, modified or unmodified, you must grant others the same rights to distribute and modify it as in the original version. By distributing it as free software under the GPLv3 license, the developers of MRST have made sure that the software will stay free, no matter who changes or distributes it.

The development of the toolbox has to a large extent been funded by strategic research grants awarded from the Research Council of Norway. Dissemination of research results is an important evaluation criterion for these types of research grants. To provide the developers with an overview of some usage statistics for the software, we ask you to kindly register your affiliation/country upon download. This information is only used when reporting impact of the creative work to agencies co-funding its development. If you also leave an email address, we will notify you when a new release or critical bug fixes are available. Your email address will under no circumstances be shared with any third party.

If you use MRST in any scholar work, we require that the creative work of the MRST developers is courteously and properly acknowledged by referring to the MRST website and by citing this book or one of the overview papers describing the software [192, 170, 30]. Last, but not least, if you have modified parts of the software or added new functionality, I strongly encourage you to release your code and thus pay back to the community that has developed it; if not the whole code, then at least generic parts that could be of significant interest to others.

COMPUTER EXERCISES

A.1.1 Download and install the software

A.1.2 Run `flowSolverTutorial1` from the command line to verify that your installation is working.

A.1.3 Load `flowSolverTutorial1` in the editor (`editflowSolverTutorial1.m`) and run it in cell mode to evaluate one cell at the time. Use `help` or `doc` to inspect the documentation for the various functions used in the script.

A.1.4 Run `flowSolverTutorial1` line by line: Set a breakpoint on the first executable line by clicking on the small symbol next to line 27, push the “play” button, and then use the “step” button to advance a single line at the time. Change the grid size to $10 \times 10 \times 25$ and rerun the example.

[5] See www.gnu.org/licenses/gpl.html for more details.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

A.1.5 Use `mrstExploreModules()` to locate and load `incompIntro` from the `incomp` module. Run the tutorial line by line or cell by cell, and then publish the workbook to reproduce the contents of Figure A.3.

A.1.6 Replace the constant permeability in the `incompIntro` tutorial by a random permeability field

```
rock.perm = logNormLayers(G.cartDims,[100 10 100])*milli*darcy;
```

Can you explain the changes in the pressure field?

A.1.7 Run all of the examples listed by `mrstExamples()` that have the word "tutorial" in their names. In particular,

- `gridTutorialIntro` introduces you quickly to the most fundamental parts of MRST, the grid structure, which is discussed in more detail in Chapter 3;
- `tutorialPlotting` introduces you to various basic routines and techniques for plotting grids and data defined over these;
- `tutorialBasicObjects` will give you a quick overview of a lot of the functionality that can be found in the toolbox.

A.2 Public Data Sets and Test Cases

Good data sets are in our experience essential to enable tests of new computational methods in a realistic and relevant setting. Such data sets are hard to come by, and when making MRST we have made an effort to provide simple access to a number of public data sets that can be freely downloaded. With the exception of a few illustrations in Chapter 3, which are based on data that cannot be publicly disclosed, all examples discussed in the book either use MRST to create their input data or rely on public data sets that can be downloaded freely from the internet.

To simplify dataset management, MRST offers a graphical user interface,

```
mrstDatasetGUI()
```

that lists all public data sets known to the software, gives a short description of each, and can download and unpack most data sets to the correct subfolder of the standard path. A few data sets require you to register your email address or fill in a license form, and in these cases we provide a link to the correct webpage. Figure A.6 shows some of the data sets that are available via the graphical interface. Several of these are used throughout the book.

Herein, I use the convention that data sets are stored in subfolders of a standard path, which you can retrieve by issuing the query

```
mrstDataDirectory()
```

I recommend that you adhere to this convention when using the software as a supplement to the book. If you insist on placing standard data sets elsewhere, I suggest that you use `mrstDataDirectory(<path>)` to reset the default path.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

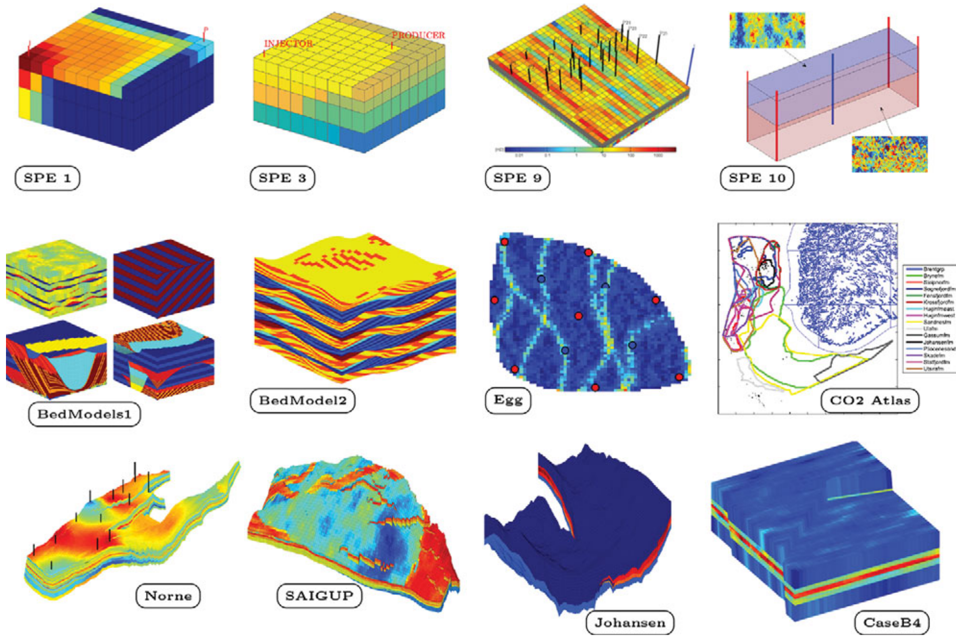

Figure A.6 Examples of the free data sets that are distributed along with MRST.

MRST also contains a number of grid factory routines and simplified geostatistical algorithms that you can use to make your own test cases. These will be discussed in more detail in Chapters 3 and 2, respectively. Here, a word of caution about exact reproducibility is in order. The grid-factory routines are mostly deterministic and should enable you to create the exact same grids each time your run them (and hence reproduce the test cases discussed in the book). Routines for generating petrophysical data rely on random numbers and will not give the same results in repeated runs. Hence, you can only expect to reproduce plots and numbers that are qualitatively similar whenever these are used, unless you make sure to store and reset the seed used by MATLAB's random number generator.

A.3 More About Modules and Advanced Functionality

As you will recall from Section A.1.1, MRST consists of core functionality and a set of add-on modules that extend, complement, and override basic features, typically in the form of specialized or more advanced solvers and workflow tools like upscaling, grid coarsening, etc. This section explains how to load and manage different modules, and tries to explain the basic characteristics of modules, or in other words the design criteria you can apply if you intend to develop your own module. For completeness, we also provide a brief overview of the comprehensive set of modules that currently are in the official release. Most of these modules are not discussed at all in the book, which for brevity needs to focus on basic flow models and modules offering standard functionality of a general interest to a wide audience.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

A.3.1 Operating the Module System

The module system is a simple facility for extending and modifying the feature set. Specifically, the module system enables on-demand activation and deactivation of advanced features that are not part of the core functionality. It consists of two parts: one that handles mapping of module names to system paths and one that uses this mapping to manipulate MATLAB's search path. The search path is a list of folders used by MATLAB to locate files. A module in MRST is strictly speaking a collection of functions, object declarations, and example scripts located in a folder. Each module needs to have a unique name and reside in a different folder than other modules. When you activate a module, the corresponding folder is added to the search path, and when you deactivate, the folder is removed from the search path.

The mapping between module names and paths on your computer is maintained by the function `mrstPath`. The paths are expected to be full paths to existing folders on your computer. To determine which modules are part of your current installation, you use the function as a command

```
mrstPath
```

This will list all modules the software is aware of and can activate. To activate a particular module, you use the function `mrstModule`. As an example, calling

```
mrstModule add mimetic mpfa
```

will load the two modules for consistent discretization methods discussed in Chapter 6. The `mrstModule` function has the following additional modes of operation:

```
mrstModule <list|clear|reset|gui> [module list]
```

which will list all active modules, deactivate all modules, deactivate all modules except for those in the list, or bring up a graphical user interface with check-boxes that enable you to activate and deactivate individual modules. For the latter, you can also use the command `moduleGUI`.

All modules that come bundled with the official release will be placed in a predefined folder structure, so that they can be added automatically to the module mapping by the `startup` function. However, module folders can in principle be placed anywhere on your computer as long as they are readable. To make MRST aware of these additional modules, you must use `mrstPath` to register them in the mapping. Assuming, for instance, that you want to make the AGMG multigrid solver [238] available. This can be done as follows,

```
mrstPath register AGMG S:\mrst\modules\3rdparty\agmg
```

Once the mapping is established, the module can be activated

```
mrstModule add AGMG
```

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

In my experience, the best way to register modules that are not part of the default mapping that comes with the official versions of the software is to add appropriate lines to your `startup_user` file. The following is an excerpt from mine

```
mrstPath reregister distmesh ...
  /home/kalie/matlab/mrst-bitbucket/mrst-core/utils/3rdparty/distmesh
```

which adds the mesh generator `DistMesh` [248] as a module in MRST.

A.3.2 What Characterizes a Module?

MRST does not have a strict policy for what becomes a module and what does not, and if you look at the modules that are part of the official release, you will see that they differ quite a lot. Some modules are just small pieces of code that add specialized capabilities like the `mpfa` module, whereas others add comprehensive new functionality such as the `incomp`, `ad-core`/`ad-blackoil`, and `co2lab` modules. Some of these modules are robust, well-documented, and contain features that will likely not change in future releases. As such, they could have been included as part of the core functionality if we had not decided to keep this as small as possible to simplify maintenance and reduce the potential for feature conflicts. Other modules are constantly changing to support ongoing research. Until recently, all modules in the AD-OO family were of this type.

Semi-independent modules is a simple way to organize software development that promotes software reuse. By organizing a new development as a separate module you will probably be more careful to distinguish code of generic value from code that is case specific or of temporary value only. The fact that others, or your future self, may want to reuse your code can motivate the extra effort needed to document and make examples and tutorials, whose presence often is decisive when people consider to use or continue to develop the functionality you have implemented. The module concept is particularly convenient if you have specific functionality you want to activate or deactivate as you like. In-house, our team has many modules that are in varying degrees of development and/or decay. Some are accessible to the whole team, whereas others reside on one person's computer only.

You may say that any module that is part of the official release is just some code that has been organized in a certain way and released to others. However, common for all such modules is that they attempt to adhere to the following recommended design rules:

- A module should offer new functionality that is distinctly different from what is already available in the core functionality and/or in other modules.
- A module should distinguish between functionality exposed to the users of the module and functionality that is only used internally. The latter can be put in a folder called

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

`private` (so that it is not accessible to functions other than those in the parent folder) or some other folder that signals that this functionality is for internal use.

- A module should contain a set of tutorials/examples that explain and highlight basic functionality of the module. The examples should be as self-explanatory as possible and at most take a few minutes to run through. Examples are easier to comprehend if you section your scripts using cell mode and accompany each code section by an informative text that explains the computations to take place or discusses visual output. If special data sets are required, these should be published along with the module.
- All main routines in a module should be documented, preferably following the format used elsewhere in the software, describing input and output data, the underlying method, and assumptions and limitations.
- Modules should, as a general rule, not use functionality from the official toolboxes that are sold with MATLAB since many users do not have access to these.
- Name conflicts should be minimized to avoid messing up the search path in MATLAB. When two files with the same name appear on the search path, MATLAB will pick the one found nearest the top. To avoid potential unintended side effects, it is therefore important that files have unique names across different modules.
- To the extent possible, the implementation should try to stick to the same naming conventions as used in MRST for readability. This means using `camelCaseNames` for functions and `CamelCase` starting with a capital letter for class objects. Likewise, widely used data objects can preferably use easily recognizable names like `G` for grid, `rock` for petrophysics, `fluid` for fluid models, `W` for wells, etc.
- Preferably, the source code should be kept in a public (or private) repository on a centralized service like Bitbucket or GitHub so that you can use a version control system to keep track of its development.

I have recently written a paper [186] that discusses development of open-source software and good practices for experimental scientific programming in more detail.

A.3.3 List of Modules

This section outlines the various modules that make up the official MRST 2018a release and explains briefly the purpose and key features of each individual module.

Grid Generation and Partitioning

The core functionality of MRST includes a general data structure for fully unstructured grids, as well as a number of grid factory and utility routines, including the possibility to read and construct corner-point grids from ECLIPSE input. In addition, there are a number of add-on modules for generating and processing grids:

`agglom`: Offers a number of elementary routines that can be combined in various ways to create coarse partitions that adapt to geology or flow fields [125, 124, 194, 187]

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

based on cell-based indicator values. More details are given in Chapter 14 and in the additional tutorial examples.

`coarsegrid`: Extends the unstructured grid format in MRST to also include coarse grids formed as a partition of an underlying fine grid. Such grids form a key part in upscaling and multiscale methods, but act almost like any standard MRST grid and can hence be passed to many solvers in other modules. Coarse grid generated from a partition are also useful for visualization purposes. More details are given in Chapter 14.

`libgeometry`: Geometric quantities like cell volumes and centroids and areas, centroids, and normals of faces must be computed by the `computeGeometry` routine. This module offers a C-accelerated version, `mcomputeGeometry`, that reduces the computational time substantially for large models. Since 2016a, the need for this C-acceleration has diminished.

`opm_gridprocessing`: Corner-point grids can be constructed from ECLIPSE input by the `processGRDECL` function, which is part of the core functionality in MRST. This module offers a C-accelerated version, `mprocessGRDECL` or `processgrid`, of the same processing routines. The implementation is from the Open Porous Media (OPM) initiative, which generally can be seen as a C/C++ sibling of MRST.

`triangle`: Provides a mex interface to `Triangle`, a two-dimensional quality mesh generator and Delaunay triangulator developed by Jonathan Richard Shewchuk, to generate grids for MRST.

`upr`: Contains functionality for generating unstructured polyhedral grids that align to prescribed geometric objects. Control-point alignment of cell centroids is introduced to accurately represent horizontal and multilateral wells, but can also be used to create volumetric representations of fracture networks. Boundary alignment of cell faces is introduced to accurately preserve geological features such as layers, fractures, faults, and/or pinchouts. This third-party module was originally developed as part of a master thesis by Berge [42]; see also [169].

Incompressible Discretizations and Solvers

This family of modules consist primarily of functionality for solving Poisson-type pressure equations:

`incomp`: Implements fluid objects for incompressible, two-phase flow. The module implements the two-point flux-approximation (TPFA) method and explicit and implicit transport solvers with single-point upstream mobility weighting, as described in detail in Parts II and III of the book. Originally part of the core module, but was moved to a separate module once the software started offering solvers for more advanced fluid models.

`mimetic`: The standard TPFA method in `incomp` is not consistent and may give significant grid-orientation errors for grids that are not K-orthogonal. The module

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

implements a family of consistent, mimetic finite difference methods for incompressible (Poisson-type) pressure equations; see Section 6.4.

mpfa: Implements the MPFA-O scheme (see Section 6.4 on page 188), which is an example of a scheme that employs more degrees of freedom when constructing the discrete fluxes across cell interfaces to ensure a consistent discretization with reduced grid-orientation effects.

vem: Virtual element methods (VEM) [38, 39] constitute a unified framework for higher-order methods on general polygonal and polyhedral grids. The module solves general incompressible flow problems by first- and second-order VEM, with the possibility to choose different inner products. Originally developed by Klemetsdal as part of his master thesis [168].

adjoint: Implements strategies for production optimization based on adjoint formulations for incompressible, two-phase flow. Example: optimization of the net present value constrained by the bottom-hole pressure in wells. For optimization problems with more complex fluid physics, the newer `optimization` module from the AD-OO framework is recommended.

Implicit Solvers Based on Automatic Differentiation

The object-oriented AD-OO framework is based on automatic differentiation and offers a rich set of functionality for solving a wide class of model equations:

ad-core: does not contain any complete simulators by itself, but rather implements the common framework used for many other modules. This includes abstract classes for reservoir models, for time stepping, nonlinear solvers, linear solvers, functionality for variable updating/switching, routines for plotting well responses, etc. See the discussion in Chapters 11 and 12 and [170, 30] for more details.

deckformat: Contains functionality for handling complete simulation decks in the ECLIPSE format, including input reading, conversion to SI units, and construction of MRST objects for grids, fluids, rock properties, and wells. Functionality from this module is essential for the fully implicit simulators in the AD-OO framework; see Chapters 11 and 12.

ad-props: Functionality related to property calculations for the AD-OO framework. Specifically, the module implements a variety of test fluids and functions that are used to create fluids from external data sets. This module is used as part of almost all simulators in MRST studying compressible equations of black-oil or compositional type. More details are in Section 11.3.

ad-blackoil: Models and examples that extend the MRST AD-OO framework found in the `ad-core` module to black-oil problems. Solvers and functionality from this module are discussed at length in Chapters 11 and 12.

blackoil-sequential: Implements sequential solvers for the same set of equations as in `ad-blackoil` based on a fractional flow formulation wherein pressure and transport are solved as separate steps; see [215] for more details. These solvers can be

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

significantly faster than those found in `ad-blackoil` for many problems, especially problems where the total velocity changes slowly during the simulation.

ad-eor: Builds on the `ad-core` and `ad-blackoil` modules and defines model equations and provides simulation tools for water-based enhanced oil recovery techniques (polymer and surfactant injection); see [30] for more details about the polymer case.

compositional: Introduced in release 2017b and offers solvers for compositional flow problems with cubic equations of state or tabulated K-values. Both the natural variables and overall composition formulations are included. More details about the underlying models and equations can be found in the PhD thesis by Møyner [212]. The tutorials include validation against commercial and external research simulators.

solvent: Introduced in release 2017b and offers an extension of the Todd–Longstaff model for miscible flooding [289] that makes it possible to simulate miscible and partially miscible displacement without using an expensive fully compositional formulation.

ad-mechanics: Introduced in release 2017b and includes a mechanical model for linear elasticity that can be coupled to standard reservoir flow models, such as oil-water or black-oil models. The global system is either solved simultaneously or by fixed-stress splitting [22]. The solver for linear elasticity uses the virtual element method and can handle general grids.

optimization: Routines for solving optimal control problems with (forward and adjoint) solvers based on the AD-OO framework. The module contains a quasi-Newton optimization routine using BFGS-updated Hessians, but can easily be set up to use any (non-MRST) optimization code.

ad-fi: Deprecated module containing our first implementation of AD-based solvers for black-oil models.

Workflow Tools

The next category of modules contains functionality that can be used either before or after a reservoir simulation as part of a general modeling workflow.

diagnostics: Flow diagnostics tools are run to establish volumetric connections and communication patterns in the reservoir and measure the heterogeneity of dynamic flow paths. Details are discussed in Chapter 13 and in [218, 261, 171].

upscaling: Implements methods and tutorials for averaging and flow-based upscaling of permeabilities and transmissibilities as discussed in Chapter 15.

steady-state: Functionality for upscaling relative permeabilities based on a steady-state assumption. This includes general steady state, as well as capillary and viscous-dominated limits. The functionality is demonstrated through a few tutorial examples. For more details, see [132, 131].

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

Multiscale Methods

Multiscale methods can either be used as a robust upscaling method to produce upscaled flow velocities and pressures on a coarse grid or as an approximate, iterative fine-scale solver. Instead of placing these modules into one of the categories discussed so far, I have given them a separate category, in part because of the prominent place they have played as a driving force for the development of MRST:

msmfe: Implements the multiscale mixed finite-element (MsMFE) method on stratigraphic and unstructured grids in 3D [68, 1, 4, 5, 6, 165, 222, 17, 194, 243]. Basis functions (prolongation operators) associated with the faces of a coarse grid are computed numerically by solving localized flow problems on pairs of coarse blocks. With these, you can define a reduced flow problem on a coarse grid and prolongate the resulting coarse solution back to the fine grid to get a mass-conservative flux field. Gives a good approximate solver and a robust and accurate alternative to upscaling for incompressible flow problems. MRST was originally developed solely to support research on MsMFE methods, but the module has nevertheless not been actively developed since 2012.

msfv: Implements the operator formulation [200] of the multiscale finite-volume (MsFV) method [146] for incompressible flow on structured and unstructured grids in 3D, under certain restrictions on the grid geometry and topology [211, 214]. Prolongation operators are defined by solving flow problems localized to dual coarse blocks and then used to define a reduced flow problem on the coarse primal grid and map unknowns computed on this grid back to the underlying fine grid. MsFV can either be used as an approximate solver or as a global preconditioner in an iterative solver. The module is not actively developed and is offered mostly for historic reasons. You should use the `msrsb` module instead.

msrsb: The multiscale restricted-smoothing basis (MsRSB) method [216, 215] is the current state-of-the-art within multiscale methods [195]. MsRSB is very robust and versatile and can either be used as an approximate coarse-scale solver that has mass-conservative subscale resolution, or as an iterative fine-scale solver that will provide mass-conservative solutions for any given tolerance. Performs well on incompressible 2-phase flow [216], compressible 2- and 3-phase black-oil type models [215, 130], as well as compositional models [217, 212]. MsRSB can utilize combinations of specialized and adapted prolongation operators to accelerate convergence [191]. In active research, and a more comprehensive version exists in-house.

Specialized Simulation Tools

This category consists of modules that implement solvers and simulators for other mathematical models than those discussed in this book.

co2lab: Combines results of more than one decade of academic research and development on CO_2 storage modeling into a unified toolchain that is easy and intuitive to

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

use. The module is geared towards the study of long-term trapping in large-scale saline aquifers and offers computational methods and graphical user interfaces to visualize migration paths and compute upper theoretical bounds on structural, residual, and solubility trapping. It also offers efficient simulators based on a vertical-equilibrium formulation to analyze pressure build-up and plume migration and compute detailed trapping inventories for realistic storage scenarios. Last but not least, the module provides simplified access to publicly available data sets, e.g., from the Norwegian CO_2 Storage Atlas. For more details, see [19] or specific references documenting the general toolchain of methods [232, 193, 21], methods for identifying structural traps [230], and the various vertical-equilibrium formulations [20, 227, 228].

`dfm`: Contains two-point and multipoint solvers for discrete fracture-matrix systems [268, 297], including multiscale methods. This third-party module is developed by Sandve from the University of Bergen, with minor modifications by Keilegavlen.

`dual-porosity`: A module for geologic well-testing in naturally fractured reservoirs has been developed and is maintained by the Carbonate Reservoir Group at Heriot Watt University. The module implements tools to generate synthetic transient pressure responses for idealized and realistic fracture networks.

`fvbiot`: Developed and maintained by the University of Bergen and implements cell-centered discretizations of three different equations: (i) scalar elliptic equations (Darcy flow), using multipoint flux approximations; this is more or less equivalent to the MPFA implementation in the `mpfa` module, although the implementation and data structures are slightly different; (ii) Linear elasticity, using the multipoint stress-approximation (MPSA) method [237, 233, 156]; (iii) Poromechanics, e.g., coupling terms for the combined system of the first two models.

`geochemistry`: Implements solvers for the following models: aqueous speciation, surface chemistry, redox chemistry, equilibrium with gas and solid phases. Each of these models can be readily coupled with a flow solver. The module is developed as a collaboration between University Texas Austin and SINTEF.

`hfm`: Implements the embedded discrete fracture method (EDFM) on stratigraphic and unstructured grids in 2D and 3D. The module also implements a multiscale restriction-smoothed basis (MsRSB) solver; see [273] for more details. The module was originally developed as a collaboration between TU Delft and SINTEF. Recently, researchers from Heriot Watt University have contributed several new functions and examples. Most notably, this includes greatly improved support for fracture shapes and orientations in 3D.

`vemmech`: Offers functionality to set up solvers for linear elasticity problems on irregular grid, using the virtual element method [39, 114], which is a generalization of finite-element methods that takes inspiration from modern mimetic finite-difference methods; see [23, 229]

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

Miscellaneous

The last category consists of modules that do not offer any computational or modeling tools but rather provide general utility functions employed by the other modules in MRST:

`book`: Contains all the scripts used for the examples, figures, and some of the exercises in this book.

`linearsolvers`: Offers bindings to external linear solvers. The initial release includes tentative support for the AMGCL header-only library for preconditioning with and without algebraic multigrid methods.

`matlab_bgl`: Routine for download the MATLAB Boost Graph Library.

`mrst-gui`: Graphical interfaces for interactive visualization of reservoir states and petrophysical data. The module includes additional routines for improved visualization (histograms, well data, etc.) as well as a few utility functions that enable you to override some of MATLAB's settings to enable faster 3D visualization (rotation, etc.).

`octave`: A number of patches to provide compatibility with GNU Octave.

`spe10`: Contains tools for downloading, converting, and loading the data into MRST; see Section 2.5.3. The module also features utility routines for extracting parts of the model, as well as a script that sets up a (crude) simulation of the full model (using the AGMG multigrid solver).

`streamlines`: Implements Pollock's method [253] for tracing of streamlines on Cartesian and curvilinear grids based on a set of input fluxes computed by the incompressible flow solvers in MRST.

`wellpaths`: Functionality for defining wells following curvilinear trajectories.

This list includes all public modules in release 2018a. By the time you read this book, more modules that are currently in the making will likely have been added to the official list.

Modules Not Part of the Official Release

The following modules are publicly available, but *not* part of the official release:

`enkf`: Ensemble Kalman filter (EnKF) module developed by researchers at TNO [182, 181] that contains EnKF and EnRML schemes, localization, inflation, asynchronous data, production and seismic data, updating of conventional and structural parameters.

`remso`: An optimization module based on multiple shooting, developed by Codas [87], which allows for great flexibility in the handling of nonlinear constraints in reservoir management optimization problems.

`mrst-cap` Implements a compositional flow model with capillary pressure; see https://github.com/sogoogos/mrst-cap.

You can also find other MRST codes online that have not been structured modules or have not been maintained for many years.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

COMPUTER EXERCISES

A.3.1 Try to run the following tutorials and examples from various modules

- `simpleBCmimetic` from the `mimetic` module.
- `simpleUpscaleExample` from the `upscaling` module
- `gravityColumnMS` from the `msmfem` module
- `example2` from the `diagnostics` module
- `firstTrappingExample` from the `co2lab` module (notice that this example does not work unless you have `MATLAB-BGL` installed).

A.4 Rapid Prototyping Using MATLAB and MRST

How can you reduce the time span from the moment you get a new idea to when you have demonstrated that it works well for realistic reservoir engineering problems? In our experience, prototyping and validating new numerical methods is painstakingly slow. There are many reasons for this. First of all, there is often a strong disconnect between the mathematical abstractions and equations used to express models and numerical algorithms and the syntax of the computer language you use to implement your algorithms. This is particularly true for compiled languages, where you typically end up spending most of your time writing and tweaking loops that perform operations on individual members of arrays or matrices. Object-oriented languages like C++ offer powerful functionality that can be used to make abstractions that are both flexible and computationally efficient and enable you to design your algorithms using high-level mathematical constructs. However, these advanced features are usually alien and unintuitive to those who do not have extensive training in computer sciences. If you are familiar with such concepts and are in the possession of a flexible framework, you still face the never-ending frustration caused by different versions of compilers and (third-party) libraries that seems to be an integral part of working with compiled languages.

Experimental Programming is Efficient in a Scripting Language

Based on working with many different students and researchers over the past twenty years, I claim that using a numerical computing environment based on a scripting language like MATLAB/GNU Octave to prototype, test, and verify new models and computational algorithms is significantly more efficient than using a compiled language like Fortran, C, and C++. Not only is the syntax intuitive and simple, but there are many mechanisms that help you boost your productivity and you avoid some of the frustrations that come with compiled languages: there is no complicated build process or handling of external libraries, and your implementation is inherently cross-platform compatible.

MATLAB, for instance, provides mathematical abstractions for vectors and matrices and built-in functions for numerical computations, data analysis, and visualization that enable you to quickly write codes that are not only compact and readable, but also efficient and

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

robust. On top of this, MRST provides additional functionality that has been developed especially for computational modeling of flow in porous media:

- an unstructured grid format that enables you to implement algorithms without knowing the specifics of the grid;
- discrete operators, mappings, and forms that are not tied to specific flow equations, and hence can be precomputed independently and used to write discretized flow equations in a very compact form that is close to the mathematical formulations of the same;
- automatic differentiation enables you to compute the values of gradients, Jacobians, and sensitivities of any programmed function without having to compute the necessary partial derivatives analytically; this can, in particular, be used to automate the formulation of fully implicit discretizations of time-dependent and coupled systems of equations;
- data structures providing unified access enable you to hide specific details of constitutive laws, fluid and rock properties, drive mechanisms, etc.

This functionality is gradually introduced and described in detail in the main parts of the book.

Interactive Development of New Programs

An equally important aspect of using a numerical environment like MATLAB is that you can develop your program differently than what you would do in a compiled language. Using the interactive environment, you can interactively analyze, change and extend data objects, try out each operation, include new functionality, and build your program as you go. This feature is essential in a debugging process, when you try to understand why a given numerical method fails to produce the results you except it to give. In fact, you can easily continue to change and extend your program during a test run: the debugger enables you to run the code line by line and inspect and change variables at any point. You can also easily step back and rerun parts of the code with changed parameters that may possibly change the program flow. Since MATLAB uses dynamic type-checking, you can also add new behavior and data members while executing a program. However, how to do this in practice is difficult to teach in a textbook. Instead, you should run and modify the various examples that come with MRST and the book. We also recommend that you try to solve the computer exercises that are suggested at the end of several of the chapters and sections in the book.

Ensuring Efficiency of Your MATLAB Code

Unfortunately, all this flexibility and productivity comes at a price: it is very easy to develop programs that are not very efficient. In the book, I therefore try to implicitly teach programming concepts you can use to ensure flexibility and high efficiency of your programs. These include, in particular, efficient mechanisms for traversing data structures like vectorization, indirection maps, and logical indexing, as well as use of advanced MATLAB functions like `accumarray`, `bsxfun`, etc. Although these will be presented in the context of reservoir

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

simulation, I believe the techniques are of interest for readers working with lower-order finite-volume discretizations on general polyhedral grids.

As an illustration of the type of MATLAB programming that will be used, we can consider the following code, which generates five million random points in 3D and counts the number of points that lie inside each of the eight octants:

```
n = 5000000;
pt = randn(n,3);
I = sum(bsxfun(@times, pt>0, [1 2 4]),2)+1;
num = accumarray(I,1);
```

The third line computes the sign of the x, y, and z coordinates and maps the triplets of logical values to an integer number between 1 and 8. To count the number of points inside each octant, we use the function `accumarray` that groups elements from a data set and applies a function to each group. The default function is summation, and by setting a unit value in each element, we count the entries.

Next, let us compute the mean point inside each octant. A simple loop-based solution could be something like:

```
avg = zeros(8,3);
for j=1:size(pt,1)
   quad = sum((pt(j,:)>0).*[1 2 4])+1;
   avg(quad,:) = avg(quad,:)+pt(j,:);
end
avg = bsxfun(@rdivide, avg, num);
```

Generally, you should try to avoid explicit loops since they tend to be slow in MATLAB. On my old laptop with MATLAB R2014a, it took 0.47 seconds to count the number of points within each octant, but 24.9 seconds to compute the mean points. So let us try to be more clever and utilize vectorization. We cannot use `accumarray` since it only works for scalar values. Instead, we can build a sparse matrix that we multiply with the `pt` array to sum the coordinates of the points. The matrix has one row per octant and one column per point. Lastly, we use the indicator `I` to assign a unit value in the correct row for each column, and use `bsxfun` to divide the sum of the coordinates with the number of points inside each octant:

```
avg = bsxfun(@rdivide, sparse(I,1:n,1)*pt, accumarray(I,1));
```

On my computer this operation took 0.53 seconds, which is 50 times faster than the loop-based solution. An alternative solution is to expand each coordinate to a quadruple $(x, y, z, 1)$, multiply by the same sparse matrix, and use `bsxfun` to divide the first three columns by the fourth column to compute the average:

```
avg = sparse(I,1:n,1)*[pt, ones(n,1)];
avg = bsxfun(@rdivide, avg(:,1:end-1), avg(:,end));
```

These operations ran in 0.64 seconds. Which solution do you think is most elegant?

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

Hopefully, this simple example has inspired you to learn a bit more about efficient programming tricks if you do not already speak MATLAB fluently. MRST is generally full of tricks like this, and in the book we will occasionally show a few of them. However, if you really want to learn the tricks of the trade, the best way is to dig deep into the actual codes.

A.5 Automatic Differentiation in MRST

Automatic differentiation (AD) is a technique that exploits the fact that any function evaluation, regardless of complexity, can be broken down to a limited set of arithmetic operations ($+$, $-$, $*$, $/$, etc.) and evaluation of elementary functions like exp, sin, cos, log, and so on, that all have known derivatives. In AD, the key idea is to keep track of quantities and their derivatives simultaneously; every time an elementary operation is applied to a numerical quantity, the corresponding differential rule is applied to its derivative.

Implementing Automatic Differentiation

Consider a scalar primary variable x and a function $f = f(x)$. Their AD representations would then be the pairs $\langle x, 1\rangle$ and $\langle f, df\rangle$, where 1 is the derivative dx/dx and df is the numerical value of the derivative $f'(x)$. Accordingly, the action of the elementary operations and functions must be defined for such pairs,

$$\begin{aligned}
\langle f, df\rangle + \langle g, dg\rangle &= \langle f + g, df + dg\rangle\,, \\
\langle f, df\rangle * \langle g, dg\rangle &= \langle fg, f\,dg + df\,g\rangle\,, \\
\exp(\langle f, df\rangle) &= \langle \exp(f), \exp(f)df\rangle\,.
\end{aligned}$$

In addition to this, we need to use the chain rule to accumulate derivatives; that is, if $f(x) = g(h(x))$, then $f_x(x) = g'(h(x))h'(x)$. If we have a function $f(x, y)$ of two primary variables x and y, their AD representations are $\langle x, 1, 0\rangle$, $\langle y, 0, 1\rangle$ and $\left\langle f, f_x, f_y\right\rangle$. This more or less summarizes the key idea behind AD; the remaining and difficult part is how to implement the idea as efficient computer code that has a low user-threshold and minimal computational overhead.

It is not very difficult to implement elementary rules needed to differentiate the function evaluations you find in a typical numerical PDE solver. To be useful, however, these rules should not be implemented as new functions, so that you need to write something like `myPlus(a, myTimes(b,c))` when you want to evaluate $a + bc$. You can find many different AD libraries for MATLAB that offer both forward and reverse model for accumulating derivatives, e.g., ADiMat [286, 43], ADMAT [60, 300], MAD [291, 275, 112], or from MATLAB Central [110, 208]. An elegant solution is to use classes and operator overloading. When MATLAB encounters an expression `a+b`, the software will choose one out of several different addition functions depending on the data types of `a` and `b`. All we now have to do is introduce new addition functions for the various classes of data types that `a` and `b` may belong to. Neidinger [223] gives a nice introduction to how to implement this

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

in MATLAB. MRST has its own implementation that is specially targeted at solving flow equations, i.e., working with long vectors and large sparse matrices.

The ADI Class in MRST

The ADI class in MRST relies on operator overloading as suggested in [223] and uses a relatively simple forward accumulation of derivatives. The ADI class differs from many other libraries in a subtle, but important way. Instead of working with a single Jacobian of the full discrete system as one matrix, *MRST uses a list of matrices that represent the derivatives with respect to different variables that will constitute sub-blocks in the Jacobian of the full system*. The reason for this choice is computational performance and user utility. In a typical simulation, we need to compute the Jacobian of systems of equations that depend on primary variables that each is a long vector. Oftentimes, we want to manipulate parts of the full Jacobian that represents specific subequations. This is not practical if the Jacobian of the system is represented as a single matrix; manipulating subsets of large sparse matrices is currently not very efficient in MATLAB, and keeping track of the necessary index sets can also be quite cumbersome from a user's point-of-view. Accordingly, our current choice is to let the MRST ADI class represent the derivatives of different primary variable as a list of matrices.

The following example illustrates how the ADI class works:

Example A.5.1 *We want to compute the expression $z = 3e^{-xy}$ and its partial derivatives $\partial z/\partial x$ and $\partial z/\partial y$ for the values $x = 1$ and $y = 2$. Using our previous notation, the AD-representation of z should be an object of the following form*

$$z = \langle 3e^{-xy}, -3ye^{-xy}, -3xe^{-xy} \rangle \approx \langle 0.4060, -0.8120, -0.4060 \rangle .$$

With the `ADI` *class, computing z and its partial derivatives is done as follows:*

```
[x,y] = initVariablesADI(1,2);
z = 3*exp(-x*y)
```

The first line tells MRST that `x` *and* `y` *are independent variables and instantiates two class objects, initialized with correct values. The second line is what you normally would write in MATLAB to evaluate the given expression. After the second line is executed, you have three ADI variables (pairs of values and derivatives):*

```
x = ADI Properties:  y = ADI Properties:   z = ADI Properties:
    val: 1               val: 2                val: 0.4060
    jac: {[1]   [0]}     jac: {[0]   [1]}      jac: {[-0.8120] [-0.4060]}
```

$\frac{\partial x}{\partial x}$ $\frac{\partial x}{\partial y}$ $\frac{\partial y}{\partial x}$ $\frac{\partial y}{\partial y}$ $\left.\frac{\partial z}{\partial x}\right|_{x=1,\,y=2}$ $\left.\frac{\partial z}{\partial y}\right|_{x=1,\,y=2}$

If we continue computing with these variables, each new computation will give a result that contains the value of the computation as well as the derivatives with respect to x and y.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

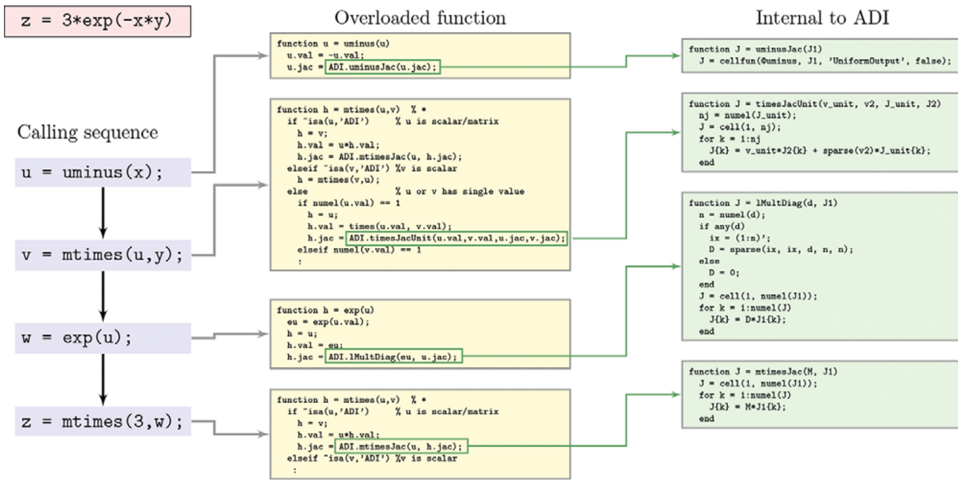

Figure A.7 Complete set of functions invoked to evaluate `3*exp(-x*y)` when `x` and `y` are ADI variables. For brevity, we have abbreviated parts of the called functions.

Let us look a bit in detail on what happens behind the curtain. We start by observing that the function evaluation $3*exp(-x*y)$ in reality consists of a sequence of elementary operations: $-$, $*$, exp, and $*$, executed in that order. In MATLAB, this corresponds to the following sequence of call to elementary functions

```
u = uminus(x);
v = mtimes(u,y);
w = exp(u);
z = mtimes(3,w);
```

To see this, you can enter the command into a file, set a breakpoint in front of the assignment to `z`*, and use the "Step in" button to step through all details. The ADI class overloads these three functions (*`uminus`*,* `mtimes`*, and* `exp`*) by new functions that have the same names, but operate on an ADI class object for* `uminus` *and* `exp`*, and on two ADI class objects or a combination of a double and an ADI class object for* `mtimes`*. Figure A.7 gives a breakdown of the sequence of calls to overloaded and internal functions that are invoked within the ADI class library to evaluate $3*exp(-x*y)$ when x and y are ADI variables. The same principles apply for other functions and if x and/or y are vectors, except that the specific sequence of overloaded and internal functions from ADI will be different.*

Computational Efficiency

As you can see from the example, use of AD will give rise to a whole new set of function calls that are not executed if you only evaluate a mathematical expression and do not compute its derivatives. Apart from the cost of the extra code lines that are executed, user-defined classes are fairly new in MATLAB and there is still some overhead in using

class objects and accessing their properties (e.g., `val` and `jac`) compared to the built-in `struct`-class. The reason why using the ADI class library still pays off in most examples, is that the cost of generating derivatives is typically much smaller than the cost of the solution algorithms they will be used in, in particular when working with equations systems consisting of large sparse matrices with more than one row per cell in the computational grid. You should nevertheless still seek to limit the number of calls involving ADI class functions (including the constructor). We let the following example be a reminder that vectorization is of particular importance when using ADI classes in MRST:

Example A.5.2 *To investigate the efficiency of vectorization versus serial execution of the ADI objects in MRST, we consider the inner product of two vectors*

```
z = x.*y;
```

and compare the cost of computing z and its two partial derivatives using four different approaches:

1. *explicit expressions for the derivative, $z_x = y$ and $z_y = x$, evaluated using standard MATLAB vectors of doubles;*
2. *the overloaded vector multiply (`.*`) with ADI vectors for x and y;*
3. *a loop over all vector elements with matrix multiply (`*=mtimes`) and x and y represented as scalar ADI variables; and*
4. *same as 3, but with vector multiply (`.*=times`).*

MATLAB offers a stopwatch timer, which we start by the command `tic` and whose elapsed time we read by the command `toc`:

```
[n,t1,t2,t3,t4] = deal(zeros(m,1));
for i = 1:m
    n(i) = 2^(i-1);
    xv = rand(n(i),1); yv=rand(n(i),1);
    [x,y] = initVariablesADI(xv,yv);
    tic, z = xv.*yv; zx=yv; zy = xv;          t1(i)=toc;
    tic, z = x.*y;                            t2(i)=toc;
    tic, for k =1:n(i), z(k)=x(k)*y(k); end;  t3(i)=toc;
    tic, for k =1:n(i), z(k)=x(k).*y(k); end; t4(i)=toc;
end
```

Figure A.8 reports the corresponding runtimes as function of the number elements in the vector. For this simple function, using ADI is a factor 20–40 times more expensive than using direct evaluation of z and the exact expressions for z_x and z_y. Using a loop will on average be more than three orders more expensive than vectorization. Since the inner iterations multiply scalars, many programmers would implement it using matrix multiply (`*`) without a second thought. Replacing (`*`) by vector multiply (`.*`) reduces the cost significantly for short vectors, but the reduction diminishes as the length of the vectors increases.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

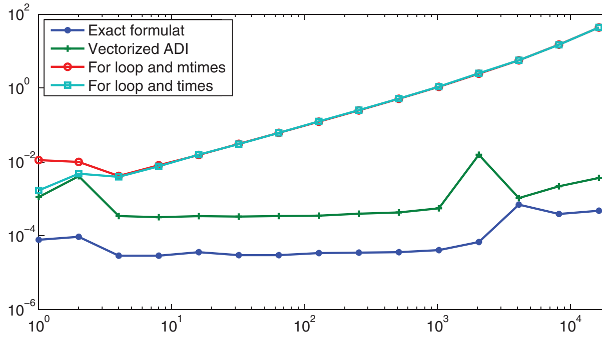

Figure A.8 Comparison of the time required for computing `z=x.*y` and derivatives as function of the number of elements in the vectors `x` and `y`.

Vectorization is usually significantly more efficient than loops when built-in or user-defined functions can be applied to complete vectors. The benefit of vectorization can be significantly diminished and completely disappear if the vectorized operations must allocate a lot of temporary memory and does not match the processor cache. Still, I would recommend that you try to vectorize as much of your code as possible, as this typically will lead to more readable code. As a general rule, you should always try to avoid loops that call functions that have non-negligible overhead. The ADI class in MRST has been designed to exploit vectorization on long vectors and lists of (sparse) Jacobian matrices and has not been optimized for scalar variables. As a result, there is considerable overhead when working with small ADI objects.

Using ADI to Assemble Linear(ized) Systems

The main use of ADI objects in MRST is to linearize and assemble (large) systems of discrete equations. To use ADI to assemble and solve a linear system $\mathbf{A}\boldsymbol{x} = \boldsymbol{b}$, we must first write the system on residual form,

$$\boldsymbol{r}(\boldsymbol{x}) = \mathbf{A}\boldsymbol{x} - \boldsymbol{b} = \mathbf{0}. \tag{A.1}$$

Since $\boldsymbol{x}$ is unknown, we assume that we have an initial guess called $\boldsymbol{y}$. Inserting this into (A.1), we obtain

$$\boldsymbol{r}(\boldsymbol{y}) = \mathbf{A}\boldsymbol{y} - \boldsymbol{b} = \mathbf{A}(\boldsymbol{y} - \boldsymbol{x}),$$

which we can solve for $\boldsymbol{x}$ to obtain $\boldsymbol{x} = \boldsymbol{y} - \mathbf{A}^{-1}\boldsymbol{r}(\boldsymbol{y})$. It follows from (A.1) that $\boldsymbol{r}'(\boldsymbol{x}) = \mathbf{A}$, which means that if we write a code that evaluates equations on residual form, we can use AD to assemble the system.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

Example A.5.3 *As a very simple illustration of how AD can be used to assemble a linear system, let us consider the following linear* 3×3 *system*

$$\begin{bmatrix} 3 & 2 & -4 \\ 1 & -1 & 2 \\ -2 & -2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -5 \\ -1 \\ 6 \end{bmatrix},$$

whose solution $x = [1\ 2\ 3]^T$ *can be computed by a single line in MATLAB:*

```
x = [3, 2, -4; 1, -4, 2; -2,- 2, 4]\ [-5; -1; 6]
```

To solve the same system using AD,

```
x  = initVariablesADI(zeros(3,1));
A  = [3,  2, -4; 1, -4,  2; -2,- 2.  4];
eq = A*x + [5; 1; -6];
u  = -eq.jac{1}\eq.val
```

or we can specify the residual equations line by line and then concatenate them to assemble the whole matrix

```
x   = initVariablesADI(zeros(3,1));
eq1 = [ 3,  2, -4]*x + 5;
eq2 = [ 1, -4,  2]*x + 1;
eq3 = [-2, -2,  4]*x - 6;
eq  = cat(eq1,eq2,eq3);
u   = -eq.jac{1}\eq.val
```

Here, the first line sets up the AD variable `x` *and initializes it to zero. The next three lines evaluate the equations that make up our linear system. Evaluating each equation results in a scalar residual value* `eq1.val` *and a* 1×3 *Jacobian matrix* `eq1.jac`. *In the fifth line, the residuals are concatenated to form a vector and the Jacobians are assembled into the full Jacobian matrix of the overall system.*

At this point, you may argue that what I have shown you is a convoluted and expensive way of setting up and solving a simple system. However, now comes the interesting part. When solving partial differential equations on complex grids, it is often much simpler to evaluate the residual equations in each grid cell than assembling the same local equations into a global system matrix. Using AD, you can focus on the former and avoid the latter. In Section 4.4.2, we introduce discrete divergence and gradient operators, `div` and `grad`. With these, discretization of the flow equation $\nabla \cdot \mathbf{K}\mu^{-1}(\nabla p - g\rho\nabla z) = 0$ introduced in Section 1.4 can be written in a form that strongly resembles its continuous form

```
eq = div((T/mu)*.(grad(p) - g*rho*grad(z)));
```

where `T` is the transmissibilities (which can be precomputed for a given grid), `mu` and `rho` are constant fluid viscosity and density, `g` is the gravity constant, and `z` is the vector of cell centroids. That single line is all we need to evaluate and assemble the corresponding linear system of discrete flow equations.

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

Our primary use of AD is for compressible flow models, which typically give large systems of nonlinear discrete equations that need to be linearized and solved using a Newton–Raphson method. As a precursor to the discussion in Chapter 7, we consider a last example to outline how you can use ADI to solve a system of nonlinear equations.

Example A.5.4 *Minimizing the Rosenbrock equation*

$$f(x,y) = (a-x)^2 + b\big(y-x^2\big)^2 \tag{A.2}$$

is a classical test problem from optimization. This problem is often called Rosenbrock's valley or banana function, since the global minimum (a, a^2) *is located inside a long, narrow and relatively flat valley of parabolic shape. Finding this valley is straightforward, but it is more challenging to converge to the global minimum. A necessary condition for a global minimum is that* $\nabla f(x,y) = 0$*, which translates to the following two equations for (A.2)*

$$\boldsymbol{g}(\boldsymbol{x}) = \begin{bmatrix} \partial_x f(x,y) \\ \partial_y f(x,y) \end{bmatrix} = \begin{bmatrix} -2(a-x) - 4bx\big(y-x^2\big) \\ 2b\big(y-x^2\big) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{A.3}$$

In the Newton–Raphson method, we assume an initial guess $\boldsymbol{x}$ *and seek a better approximation* $\boldsymbol{x} + \boldsymbol{\Delta x}$ *by solving for* $\boldsymbol{\Delta x}$ *from the linearized equation*

$$\boldsymbol{0} = \boldsymbol{g}(\boldsymbol{x} + \boldsymbol{\Delta x}) \approx \boldsymbol{g}(\boldsymbol{x}) + \nabla \boldsymbol{g}(\boldsymbol{x})\boldsymbol{\Delta x}. \tag{A.4}$$

This is quite simple using ADI in MRST:

```
[a, b, tol] = deal(1, 100, 1e-6);
[x0, incr]  = deal([-.5; 4]);
while norm(incr)>tol
   x = initVariablesADI(x0);
   eq = cat( 2*(a-x(1)) - 4*b.*x(1).*(x(2)-x(1).^2), ...
             2*b.*(x(2)-x(1).^2));
   incr = - eq.jac{1}\eq.val;
   x0 = x0 + incr;
end
```

This is just a backbone version of a Newton solver that does not contain safeguards of any kind like checking that the increments are finite, ensuring that the loop terminates after a finite number of iterations, etc. Figure A.9 illustrates how the Newton solver converges to the global minimum.

Beyond the examples and the discussion in this Appendix, we will not go more into details about the technical considerations that lie behind the implementation of AD in MRST. If you want a deeper understanding of how the ADI class works, the source code is fully open, so you are free to dissect the details to the level of your own choice.

Computer exercises

A.5.1 As an alternative to using AD, you can use finite differences, $f'(x) \approx [f(x+h) - f(x)]/h$, or a complex extension to compute $f'(x) \approx \mathrm{Im}(f(x+ih))/h$. Use AD and the function

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press

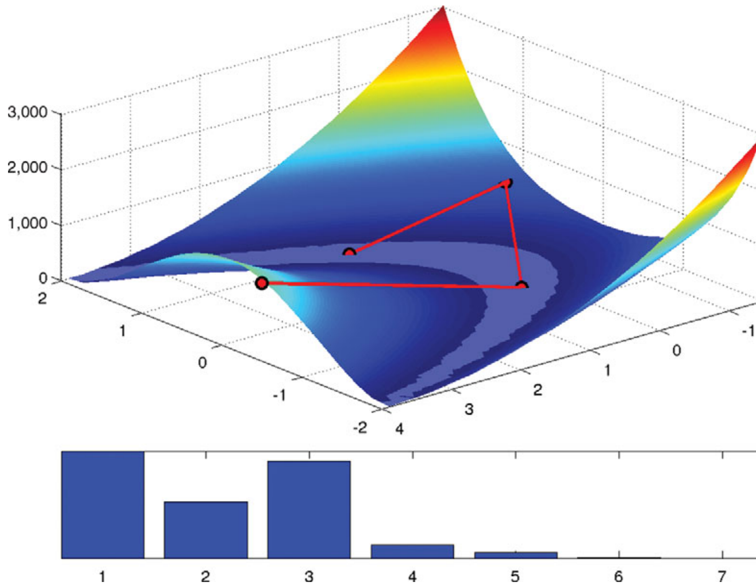

Figure A.9 Convergence of the AD-based Newton solver for the Rosenbrock problem. The upper plot shows the path taken by the nonlinear solver superimposed over the function $f(x,y)$. We have used a modified colormap to show the valley in which the function f attains its 1% lowest values. The lower plot depicts $f(x,y)^{1/10}$ for the initial guess and the six iterations needed to reduce the norm of the increment below 10^{-6}.

```
f = @(x) exp((x-.05).*(x-.4).*(x-.5).*(x-.7).*(x-.95));
```

to assess how accurate the two methods approximate $f'(x)$ at n equidistant points in the interval $x \in [0,1]$ for different values of h.

A.5.2 The ADI class can compute `log` and `exp`, but does not yet support `log2`, `log10`, and `logm`. Study `ADI.m` and see if you can implement the missing functions. What about trigonometric and hyperbolic functions?

A.5.3 Can AD be used to compute higher-order derivatives? How or why not?

https://doi.org/10.1017/9781108591416.021 Published online by Cambridge University Press