Introduction to the MATLAB Reservoir Simulation Toolbox (MRST)

Knut–Andreas Lie

Department of Mathematics and Cybernetics, SINTEF Digital/
Department of Mathematical Sciences, NTNU, Norway

Multiscale Methods Summer School
June 26–29, 2017, Hasselt, Belgium
Outline

1. Introduction
2. Getting started with MRST
3. Grids and petrophysical data
4. Incompressible flow
5. Multiphase flow
6. Compressible flow
7. The AD-OO framework in MRST
The MATLAB Reservoir Simulation Toolbox

Open-source toolbox for reservoir modelling, developed by SINTEF Digital and used in most of our research

Wide international user base:
- academic institutions, oil and service companies
- USA, Norway, China, Brazil, United Kingdom, Iran, Germany, Netherlands, France, Canada, . . .
- 10 000+ unique downloads since 2013

Used in publications:
- 24 PhD theses and 63 master theses
- 110+ journal/proceedings papers by authors outside our group

http://www.sintef.no/MRST
Toolbox for experimental programming

Flexible simulators, easy to extend with new functionality, scaling with accuracy requirement and computational budget

<table>
<thead>
<tr>
<th>seconds</th>
<th>minutes</th>
<th>hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diagnostics/proxies</td>
<td>Model reduction</td>
<td>Full simulation</td>
</tr>
<tr>
<td>Flow diagnostics/volumetrics</td>
<td>Grid coarsening</td>
<td>Black-oil, EOR, thermal, compositional, geomechanics</td>
</tr>
<tr>
<td>Physics-based proxies</td>
<td>Flow-based upscaling</td>
<td>Grids and discretizations</td>
</tr>
<tr>
<td>Fast optimization</td>
<td>Multiscale methods</td>
<td>Nonlinear/linear solvers</td>
</tr>
<tr>
<td>Spill-point analysis</td>
<td>Model-reduction techniques</td>
<td>Rapid prototyping</td>
</tr>
<tr>
<td></td>
<td>Vertical-equilibrium models</td>
<td>Adjoint formulations and (closed-loop) optimization</td>
</tr>
</tbody>
</table>
Why in MATLAB?

Prototyping in a scripting language is much less time-consuming than in traditional compiled languages (C, C++, Fortran, ...)

- Explore alternative algorithms/implementations close to mathematics
- Gradually replace individual (or bottleneck) operations with accelerated editions callable from MATLAB
- Direct access to MATLAB environment and prototype whilst developing replacement components
- MATLAB/Octave is widely used in academic institutions

If we were to start all over, we might as well have chosen Python
How the software is organized . . .

- **Add-on modules**
  - Grid coarsening
  - Flow diagnostics
  - Discretizations
  - Multiscale methods
  - Upscaling
  - Visualization
  - Co2lab
  - Input decks

**MRST core**

- Fully implicit

**CO2 saturation at 500 years**
- 16%
- 12%
- 3%
- 56%
- 12%

**Injected volume**: \(2.185 \times 10^7 \text{ m}^3\)

**Height of CO2−column**
- Residual (traps)
- Residual (plume)
- Movable (traps)
- Movable (plume)
- Leaked

**Cells.faces = faces.nodes = faces.neighbors**
- 1 10 1 1 0 2
- 1 8 1 2 2 3
- 1 7 2 1 3 5
- 2 1 2 3 5 0
- 2 2 3 1 6 2
- 2 5 3 4 0 6
- 3 3 4 1 1 3
- 3 7 4 7 4 1
- 3 2 5 2 6 4
- 4 8 5 3 1 8
- 4 12 6 2 8 5
- 4 9 6 6 4 7
- 5 3 7 3 7 8
- 5 4 7 4 0 7
- 5 11 8 3
- 6 9 8 5
- 6 6 9 3
- 6 5 9 6
- 7 13 10 4
- 7 14 10 5
- ...
The core module provides basic data structures and utility functions. Add-on modules offer:

- discretizations and solvers
- simulators for incompressible and compressible flow
- workflow tools such as coarsening, upscaling, flow diagnostics, visualization, etc
- special models like geomechanics and fractured reservoirs
- analysis of large-scale CO₂ storage in saline aquifers
Two different programming paradigms

**incomp** – sequential solvers for incompressible flow
- Have been part of MRST since the start
- Uses imperative programming: functions that operate mainly on vectors, (sparse) matrices, structures, and a few cell arrays
- Explicit assembly and linearization of flow equations

**AD-OO** – (fully) implicit solvers for compressible flow
- More recent addition to MRST
- Object-oriented framework for building simulators
- Assembly and linearization performed implicitly by the use of automatic differentiation

Both families rely on functionality from mrst-core
Quick overview of functionality

Basic functionality in MRST:

- grid structure and grid factory routines
- petrophysical data and incompressible fluid models
- physical units and conversion routines between SI and common field units
- routines for setting and manipulating boundary conditions, sources/sinks, and well models
- reservoir state (pressure, fluxes, saturations, . . .)
- visualization routines for cell and face data
Quick overview of functionality

Grid generation and coarsening:

- **MRST core**
- **upr**
- **coarsegrid**
- **agglom**
- **libgeometry**
- **opm_gridprocessing**
- **triangle**

MRST offers a wide variety of grid factory routines and input from Eclipse, ukr generates 2D and 3D Voronoi grids with cell and face constraints

coarsegrid: data structures and simple coarsening, adapted partitions in agglom
C-accelerated: processing in libgeometry and opm_processing.
incomp implements TPFA-based flow solver and explicit/implicit transport solvers; mimetic, mpfa, ntpfa, and vem implement consistent discretizations.
Quick overview of functionality

Discretization and solvers for compressible flow:

- **ad-core**
- **ad-blackoil**
- **ad-eor**
- **blackoil-sequential**
- **deckreader**
- **ad-props**

The AD-OO framework offers fully implicit simulators from industry-standard input decks, including computations of adjoints.
Quick overview of functionality

Upscaling and multiscale methods:

- **msrsb**: is state-of-the-art multiscale solver, 
- **hfm**: implements this for fracture models,
- **msmfem** and **msfvm** are earlier developments

**Upscaling**: flow-based single-phase upscaling,
**steady-state**: multiphase upscaling
Quick overview of functionality

Fractured media:

- dfm – discrete fracture models
- hfm – hierarchical/embedded fracture models
- dual-porosity

Geomechanics:

- ad-mechanics – coupled flow and mechanics (R2017b)
- vemmech – virtual element methods
- fvbiot – multipoint stress-approximation methods
Quick overview of functionality

Workflow tools:

- **co2lab**: comprehensive tools for large-scale CO₂ storage saline aquifers
- **diagnostics**: flow diagnostics, **mrst-gui**: interactive visualization
- **optimization**: solution of optimal control problems based on AD-OO
- **enkf** and **remso**: third-party modules for EnKF and multiple-shooting optimization
Outline

1. Introduction
2. Getting started with MRST
3. Grids and petrophysical data
4. Incompressible flow
5. Multiphase flow
6. Compressible flow
7. The AD-OO framework in MRST
You are here: MRST / Download

Download

SINTEF publishes several sets of resources as part of the Matlab Reservoir Simulation Toolbox. This is a list of packages and datasets currently available for download.

The MATLAB Reservoir Simulation Toolbox

MRST is a set of core features intended to assist the student, researcher and practitioner who analyses reservoir-type flows or develops numerical methods for solving flow or transport problems in reservoir applications. It is the intention of the MRST developers that the package be a solid foundation for grid handling, visualisation, and advanced discretisations.

Sources to the current as well as a few, selected previous releases of the core MRST package set are available on a separate download page. Please fill out the accompanying form to download the package.

Public Data Sets

- The SAIGUP data set is a single realisation from the Sensitivity Analysis of the Impact of Geological Uncertainties on Production project. The data is used in a number of examples accompanying the 2011a and later releases of MRST. You may download a copy of the data from the following web page.
- The Johansen formation is a candidate site for large-scale CO2 storage offshore the south-west coast of Norway. The MatMoRA project has developed a set of geological models based on available seismic and well data. You may download a copy of the data from the following web page.

Published February 23, 2011

From http://www.sintef.no/MRST
MRST is provided as self-contained archive file. The following command

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

will create a directory `mrst-2016b` in your current working directory.

Once MRST has been extracted to some directory, you must navigate MATLAB there. On Linux/Mac OS,

```
cd /home/username/mrst-2016b/
```

or on Windows,

```
cd C:\Users\username\mrst-2016b\n```

assuming that the files were extracted to the home directory. The startup.m file must then be run to activate MRST,

```
startup;
```

or you can call the startup script directly

```
run /home/username/mrst-2016b/startup
```
Getting started: welcome message

If you start MATLAB in the directory containing MRST, or run the `startup.m` file, you will see the following message:

```
>> startup
Welcome to the Matlab Reservoir Simulation Toolbox (MRST)!
You are using the release version 2016b. To download other versions of MRST and view examples and relevant publications, please visit www.sintef.no/mrst.

Useful commands for getting started:
- List all introductory examples: mrstExamples()
- List all modules: mrstPath('list')
- Load modules using GUI: mrstModule('gui')
- Explore all available data sets: mrstDatasetGUI()
- List examples of a module: mrstExamples('ad-blackoil')
- Show all examples in all modules: mrstExamples('all')
- Explore modules and publications: mrstExploreModules()
- Display this message: mrstStartupMessage()

For assistance and discussions about MRST, please visit our mailing list at www.sintef.no/projectweb/mrst/forum/ (sintef-mrst@googlegroups.com)
For some common queries, see our FAQ: www.sintef.no/projectweb/mrst/faq/
```
Getting started: sources for information

- the MRST book and key publications – the book gives a comprehensive introduction to basic flow simulation as implemented in MRST; three papers give more condensed overviews
- example scripts
- Jolts (just-in-time online learning tools)
- module examples
- manual pages for individual routines
- the source code itself
- FAQ webpage and MRST-users mailing list
- public data sets
330 10 Solvers for Incompressible Immiscible Flow

Fig. 10.13. Illustration of the sloping sandbox used for the buoyancy example and how it is simulated by rotating the gravity vector. (Color: Gaussian porosity field).

```matlab
R = makehgtform('yrotate', -pi*theta/180);
gravity reset on
gavity( R(1:3,1:3)*gravity() );
```

MRST defines the gravity vector as a persistent, global variable which by default equals $\vec{0}$. The second line ensures that $\vec{g}$ is set to the standard value (pointing downward in the vertical direction) before we perform the rotation.

To initialize the problem, we assume that CO$_2$, which is lighter than the resident brine, fills up the model from the bottom and to a prescribed height,

```matlab
xr = initResSol(G, 1*barsa, 1);
d = gravity() ./ norm(gravity);
dc = G.cells.centroids * d;
xr.s(dc>max(dc)-height) = 0;
```

For accuracy and stability, the time step is ramped up gradually as follows,

```matlab
dT = [.5, .5, 1, 1, 2, 2, 2, 5, 5, 10, 10, 15, 20, ...
    repmat(25,[1,97])].*day;
```

to reach a final simulation time of 2500 days. The remaining code is similar to what was discussed above; details can be found in `buoyancyExample.m`.

Let us consider the homogeneous case first. Initially, the buoyant CO$_2$ plume will form a cone shape as it migrates upward and gradually drains the resident brine. After approximately 175 days, the migrating plume starts to accumulate as a thin layer of pure CO$_2$ under the sloping east face of the box. This layer will migrate quickly up towards the topmost northeast corner of the box, which is reached after approximately 400 days. This corner forms a structural trap that will gradually be filled as more CO$_2$ migrates upward. The trapped CO$_2$ forms a diffused and curved interface (see the plots at 500 and 1000 days), but as time passes, the interface becomes sharper and flatter. During the same period, brine will imbibe into the trailing edge of the CO$_2$ plume and gradually formed a layer of pure brine at the bottom.
The core module of MRST offers a number of examples that introduce you to data structures and data sets, how to set up basic solvers, how to visualize input data and simulation results, etc.

>> mrstExamples
Module "core" has 18 examples:
   flowSolverTutorial1.m
   flowSolverTutorialAD.m
   tutorialAD.m
   tutorialBasicObjects.m
   tutorialPlotting.m
   datasets/showCaseB4.m
   datasets/showJohansen.m
   datasets/showNorne.m
   datasets/showSAIGUP.m
   datasets/showSPE10.m
   grids/gridTutorialCornerPoint.m
   grids/gridTutorialIntro.m
   grids/gridTutorialStruct.m
   grids/gridTutorialUnstruct.m
:
... presented in workbook format

Basic Flow-Solver Tutorial

The purpose of this example is to give an overview of how to set up and use a standard two-point pressure solver to solve the single-phase pressure equation

$$\nabla \cdot \mathbf{v} = q, \quad \mathbf{v} = -\frac{K}{H} \nabla p,$$

for a flow driven by Dirichlet and Neumann boundary conditions. Our geological model will be simple a Cartesian grid with anisotropic, homogeneous permeability.

In this tutorial example, you will learn about:

1. the grid structure,
2. how to specify rock and fluid data,
3. the structure of the data-objects used to hold solution,
4. how to assemble and solve linear systems,
5. useful routines for visualizing and interacting with the grids and simulation results.

Contents

- Define geometry
- Process geometry
- Set rock and fluid data
- Initialize reservoir simulator
- Impose Dirichlet boundary conditions
- Construct linear system
- Inspect results

Define geometry

Construct a Cartesian grid of size 10-by-10-by-4 cells, where each cell has dimension 2-by-1-by-1. Because our flow solvers are applicable for general unstructured grids, the Cartesian grid is here represented using an unstructured format, in which cells, faces, nodes, etc. are given explicitly.

\[ nx = 10; \quad ny = 10; \quad nz = 4; \]

G = cartGrid([nx, ny, nz]);

display(G);

g =

cells: [1x1 struct]
faces: [1x1 struct]
Short learning modules consisting of 3-10 minute videos covering a specific topic. 

**Jolt1**: explains what MRST is, how to download it, and how to make your first flow solvers. **Jolt2**: introduction to grid and grid generation.
Finding more information . . .

- the MRST book and key publications
- example scripts
- Jolts (just-in-time online learning tools)
- module examples – many of the modules contain example scripts located in the subdirectory ’examples’ of the module that outline functionality provided in the module. Some of these examples are available on the module overview pages
- manual pages for individual routines
- the source code itself
- FAQ webpage and MRST-users mailing list
- public data sets
Grid Coarsening

The module implements functionality for generating coarse partitions and turning these into MRST grids.

Tutorials

Example 1
This example shows you how to partition rectangular 2D Cartesian grids, the relationship between cell and block numbers, and outlines the basics of the coarse-grid structure, including numbering of cells, faces, and node.

Example 2
We show partitions of grids representing more complex domains: a rectangular grid with a semi-circular cutout, a 3D cup-formed domain, a 2D Voronoi grid of rectangular domain with a quarter-circle cutout, and a corner-point grid with a single fault.

Example 3
The example continues the discussion of the coarse-grid structure and shows how we can partition the coarse faces so that there are more than one face (connection) between neighboring coarse blocks.

Example 4
In this example, we take a closer look at partition vectors and discuss how different types of partitions can be combined into one.

Example 5
In this example, we use the function `refineNearWell` to make coarse grids with various types of near-well refinement. The examples uses both Cartesian and 2.5 D PEBI fine grids.

Example 6
We partition the Norne field uniformly in logical Cartesian space. Since the model contains many inactive cells, the initial partition must be postprocessed to ensure a contiguous partition vector. We visualize some of the coarse blocks and show how they are connected with their neighbors.
Start GUI with the command: mrstExploreModules
\[ \nabla \cdot \vec{v} = q, \quad \vec{v} = -\frac{K}{\mu} \left[ \nabla p + \rho g \nabla z \right] \]

**Vertical well and Dirichlet boundary**

```
% Grid and rock parameters
nx = 20; ny = 20; nz = 10;
G = computeGeometry(cartGrid([nx, ny, nz]));
rock.perm = repmat(100 * milli*darcy, [G.cells.num, 1]);
fluid = initSingleFluid('mu', 1*centi*poise, ...
    'rho', 1014*kilogram/meter^3);
gravity reset on

% Fluid sources and boundary conditions
c = (nx/2*ny+nx/2 : nx*ny : nx*ny*nz);!
src = addSource([], c, ones(size(c)) ./ day());
bc = pside([], G, 'LEFT', 10*barsa());

% Compute transmissibilities
T = computeTrans(G, rock);

% Solve the system and convert to bars
rSol = initState(G, [], 0);
rSol = incompTPFA(rSol, G, fluid, 'src', src, 'bc', bc);
p = convertTo(rSol.pressure, barsa());
```

From tutorial: incompTutorialSRCandBC.m
Operating modules

Graphical user interface to modules:

```python
mrstModule('gui')
moduleGUI
```

List all modules and their path

```python
mrstPath
```

Load new modules

```python
mrstModule add mimetic mpfa
```

Adding your own modules

```python
mrstPath reregister distmesh ...
/home/username/mrst–2016b/utils/3rdparty/distmesh
```
Finding more information . . .

- the MRST book and key publications
- example scripts
- Jolts (just-in-time online learning tools)
- module examples
- manual pages for individual routines – all routines in MRST are documented using a style similar to standard MATLAB that describes synopsis, input/output parameters, and how the routine works. In most cases, the documentation also offers simple examples of usage and list related routines
- the source code itself
- FAQ webpage and MRST-users mailing list
- public data sets
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

\[
\begin{bmatrix}
  k_1 & k_2 \\
  k_2 & k_3
\end{bmatrix}
\]

in two space dimensions

\[
\begin{bmatrix}
  k_1 & k_2 & k_3 \\
  k_2 & k_3 & k_4
\end{bmatrix}
\]

in three space dimensions.

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. This property is guaranteed by function 'computeGeometry'.

SEE ALSO:
computeGeometry, computeMimeticIP, darcy, permTensor.
Finding more information . . .

- the MRST book and key publications
- example scripts
- Jolts (just-in-time online learning tools)
- module examples
- manual pages for individual routines
- the source code itself – all parts of MRST are available as open source code. However, MRST is a research tool that was developed primary to provide a flexible development platform, and some parts of the software may admittedly be quite hard to digest for those unfamiliar with our way of writing efficient MATLAB
- FAQ webpage and MRST-users mailing list
- public data sets
% Vectors from cell centroids to face centroids

cellNo = rldecode(1:G.cells.num, diff(G.cells.facePos), 2);  
if ~isempty(opt.cellCenters)
    C = opt.cellCenters;
else
    C = G.cells.centroids;
end
if ~isempty(opt.cellFaceCenters)
    C = opt.cellFaceCenters - C(cellNo,:);
else
    C = G.faces.centroids(G.cells.faces(:,1), :) - C(cellNo,:);
end

% Normal vectors
sgn = 2*(cellNo == G.faces.neighbors(G.cells.faces(:,1), 1)) - 1;
N = bsxfun(@times, sgn, G.faces.normals(G.cells.faces(:,1),:));
clear sgn;

if strcmpi(opt.K_system, 'xyz'),
    [K, i, j] = permTensor(rock, G.griddim);

    assert (size(K,1) == G.cells.num, ...
        ['Permeability must be defined in active cells only.' ... 'Got %d tensors, expected %d (== number of cells).'], ...
        size(K,1), G.cells.num);

    % Compute T = C'K*N / C'C. Loop—based to limit memory use.
    T = zeros(size(cellNo));
    for k=1:size(i,2),
        T = T + sum(C(:,i(k)) .* K(cellNo,k) .* N(:,j(k)), 2);
    end
    clear K i j cellNo N;
end
T = T ./ sum(C.*C,2);
clear C;
Finding more information . . .

- the MRST book and key publications
- example scripts
- Jolts (just-in-time online learning tools)
- module examples
- manual pages for individual routines
- the source code itself
- FAQ webpage and MRST-users mailing list
- public data sets
The MRST-users mailing list

Hosted on Google group: sintef-mrst@googlegroups.com

Please consider the following points before posting a question:

- Search the forum to check if your question has been answered
- Formulate your question carefully. If we cannot understand, we cannot help you
- For code-technical issues, prepare a complete minimal example
- Please help out in answering questions from other users
- Have a little patience; this is not a 24-7 emergency line

URL: https://groups.google.com/forum/#!forum/sintef-mrst
Finding more information . . .

- the MRST book and key publications
- example scripts
- Jolts (just-in-time online learning tools)
- module examples
- manual pages for individual routines
- the source code itself
- FAQ webpage and MRST-users mailing list
- public data sets – a number of data sets are offered alongside with the software; these data sets are used in various examples and tutorials of the software
Public data sets

SPE 1  SPE 3  SPE 9  SPE 10

BedModels1  BedModel2  Egg  CO2 Atlas

Norne  SAIGUP  Johansen  CaseB4
Public data sets

Graphical user interface to download and get information about data sets:

```
mrstDatasetGUI()
```

Information about specific data set

```
getDatasetInfo('norne')
```

Path for each known data set

```
getDatasetPath('norne')
```

Query/set path for all public data sets

```
mrstDataDirectory
mrstDataDirectory('/home/username/mydata/mrst/')
```
Software requirements

Minimal requirement is MATLAB version 7.4 (R2007a).

Certain modules use features that were not present in R2007a:

- Automatic differentiation relies upon new-style classes (classdef) from R2008a.
- Various scripts and examples use new syntax for random numbers from R2007b.
- Some scripts in the modules may use tilde operator to ignore return values (e.g., $[\sim,i]=\text{max}(x,1)$) from R2009b.
- Some solvers (e.g., fully implicit) are not efficient on versions older than R2011b.

Most of MRST can be used with Octave, except for graphical user interfaces and some functionality in the object-oriented framework for fully implicit solvers based on automatic differentiation.

More information: http://www.sintef.no/Projectweb/MRST/FAQ/
Apart from MATLAB, MRST does not rely on any third-party software/libraries.

However, the following are useful:

- **AGMG** – simple but efficient algebraic multigrid solver
- **MATLAB-BGL** – MATLAB Boost Graph Library
- **METIS** – partitioning of fully unstructured grids, etc.
- **Export_fig** – to produce high quality figures for publications

More information: [http://www.sintef.no/Projectweb/MRST/FAQ/](http://www.sintef.no/Projectweb/MRST/FAQ/)
Citing MRST

If you are using MRST in any publication, we would be grateful if you cite the MRST book or one of the following three papers (possibly in addition to a link to our webpage):


An earlier version was published as: K. Bao, K.-A. Lie, O. Møyner, and M. Liu Fully-implicit simulation of polymer flooding with MRST. ECMOR XV, Amsterdam, Netherlands, 29 Aug–1 Sept, 2016. DOI: 10.3997/2214-4609.201601880


Complete MATLAB scripts that reproduce (almost) all the figures and examples in the paper are available for download, see e.g., Example 5 and Example 6.


Scientific publications utilizing MRST

MRST has been used in a large number of journal articles, conference proceedings, and master and doctoral theses. We try to collect as many as possible of these publications and have compiled them in separate lists. If your paper is missing in the lists, if have you used MRST in your thesis, or if have you supervised students using MRST, we are of course very happy to hear about it. If you provide us with publication details, we will list your publication or details about the thesis. If you also provide us with an illustrative picture and a short description, we will highlight your work on our gallery pages. Contact: Knut-Andreas.Lie@sintef.no
Computer exercises

- Download and install the software

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

- Load the `flowSolverTutorial1` tutorial in the editor and run it in cell mode. Use `help` or `doc` to inspect the documentation for the various functions that are used in the script.

- Run the `flowSolverTutorial1` tutorial 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.

- Use `mrstExploreModules()` to locate and load the `incompIntro` tutorial from the `incomp` module. Examine the tutorial in the same way as you did for `flowSolverTutorial1`. Publish the workbook.
Outline

1. Introduction
2. Getting started with MRST
3. Grids and petrophysical data
4. Incompressible flow
5. Multiphase flow
6. Compressible flow
7. The AD-OO framework in MRST
What you will learn in this section

In this section, we will discuss:

- standard grids and grid factory routines in MRST
- stratigraphic grids
- petrophysical properties and simplified geostatistics
- unstructured representation used in MRST
- techniques for manipulating (and visualizing) grids

You will also get a taste of plotting routines, common tricks, etc

To learn more:
- watch the videos in Jolt2
- study tutorials called gridTutorial*.m in mrst-core
- read Lie et al. (COMG, 2012), doi: 10.1007/s10596-011-9244-4
- read Chapters 2 and 3 in the MRST book
The fundamental object in MRST is the grid:

- all grids are assumed to be unstructured
- data structure for geometry and topology
- several grid factory routines
- input of industry-standard formats

Physical quantities defined as dynamic objects in Matlab:

- properties of medium: $\phi$, $K$, net-to-gross, …
- reservoir fluids: $\rho$, $\mu$, $k_r$, PVT, …
- driving forces: wells, boundary conditions, sources
- reservoir state: pressure, fluxes, saturations, …
- we assume SI units (e.g., $[K] = m^2$, $[\mu] = Pa \cdot s$, …)

Functions in MRST accept these objects as input, manipulate them, and produce them as output.
Standard grids: Cartesian grids

**Regular Cartesian grids:**

\[ G = \text{cartGrid}([nx, ny, Lx, Ly]); \]
\[ G = \text{cartGrid}([nx, ny, nz, Lx, Ly, Lz]); \]
\[ G = \text{cartGrid}([10, 20, 5], [5, 10, 1]); \]
\[ \text{plotGrid}(G), \text{view}(3); \text{axis equal} \]

**Rectilinear grids:**

\[ G = \text{tensorGrid}(x, y); \]
\[ G = \text{tensorGrid}(x, y, z); \]
\[ G = \text{tensorGrid}((0:.1:2), (0:.1:1).^2); \]
\[ \text{plotGrid}(G,'\text{FaceColor}', [.7 .7 1]); \]
**Standard grids: curvilinear grids**

Create a rough grid by perturbing the inner nodes randomly

\[
\begin{align*}
nx &= 12; \ ny = 6; \\
G &= \text{cartGrid}([nx, ny]); \\
\text{plotGrid}(G, '\text{LineStyle}', ':'); \\

I &= \text{true}(G.\text{nodes}.\text{num}, 1); \ % \text{Logical indexing is fast in MATLAB} \\
I( \text{gridFaceNodes}(G, \text{boundaryFaces}(G))) &= \text{false}; \\
G.\text{nodes}.\text{coords}(I,:) &= G.\text{nodes}.\text{coords}(I,:); + 0.6*(\text{rand}(\text{sum}(I), 2) - 0.5); \\
\text{plotGrid}(G, '\text{FaceColor}', 'none', '\text{LineWidth}', 1);
\end{align*}
\]
Mapping a curvilinear grid to a complex shape is generally difficult.

Alternative approach: embed the domain within a larger “fictitious” domain of simple shape, boolean indicator value tells whether each cell is part of the domain or not. Here, an ellipsoid within a cube:

```matlab
dx = linspace(-2,2,21);
G = tensorGrid(x,x,x);
subplot(1,2,1); plotGrid(G);view(3); axis equal
subplot(1,2,2); plotGrid(G,'FaceColor','none');
G = computeGeometry(G);
c = G.cells.centroids;
r = c(:,1).^2 + 0.25*c(:,2).^2+0.25*c(:,3).^2;
G = removeCells(G, r>1);
plotGrid(G); view(-70,70); axis equal;
```
Standard grids: Delaunay and Voronoi grids

For 3D tetrahedral grids, MRST supplies the function `tetrahedralGrid(x,y,z)`. The function `pebi()` has no natural counterpart in 3D, but routines exist in the `upr` module.
Using an external grid generator

Install DistMesh by Persson & Strang as a module

```python
path = fullfile(ROOTDIR,'utils','3rdparty','distmesh');
mkdir(path)
unzip('http://persson.berkeley.edu/distmesh/distmesh.zip', path);
mrstPath('rregister','distmesh', path);
```

and use it to grid around a circular inclusion

```python
mrstModule add distmesh;
fd=@(p) ddiff(drectangle(p,-1,1,-1,1), dcircle(p,0,0,0.5));
[p,t]=distmesh2d(fd, @huniform, 0.2, [-1,-1;1,1], [-1,-1;1,1,-1;1,1]);
G = triangleGrid(p, t);
```

For details: see Chapter 3.2.4 of the MRST book
Layered and stratigraphic grids

MRST has a few simple routines for generating layered/stratigraphic grids

% Extrude a standard MATLAB dataset
load seamount
G = tensorGrid(x, sqrt(0:0.05));
plotGrid(G,'FaceAlpha',0.8);
plotFaces(G,find(G.faces.tag>0), 'FaceColor','red');
view(40,40), axis off

% Make and process simple corner—point description
grdecl = simpleGrdecl([20, 10, 5], 0.12)
G = processGRDECL(grdecl);
plotGrid(G,'FaceAlpha',0.8);
plotFaces(G,find(G.faces.tag>0), 'FaceColor','red');
view(40,40), axis off
All flow and transport solvers in MRST require a rock structure, which by convention is called \texttt{rock}, and contains two fields:

- \texttt{rock.poro} – porosity, column vector with one entry per active cell
- \texttt{rock.perm} – permeability in SI units

The permeability can either be a single column (isotropic), two or three columns (diagonal tensor), or a symmetric, full tensor permeability

\[
K_i = \begin{bmatrix} K_1(i) & K_2(i) \\ K_2(i) & K_3(i) \end{bmatrix}, \quad K_i = \begin{bmatrix} K_1(i) & K_2(i) & K_3(i) \\ K_2(i) & K_4(i) & K_5(i) \\ K_3(i) & K_5(i) & K_6(i) \end{bmatrix}
\]

Nonsymmetric permeabilities are currently not supported

The \texttt{rock} object can also hold net-to-gross, \texttt{ntg}, consisting of a scalar or a single column vector with one value per active cell
Square 10 × 10 grid model with a uniform porosity of 0.2 and isotropic permeability equal 200 mD:

\[
\begin{align*}
G &= \text{cartGrid([10 10]);} \\
\text{rock} &= \text{makeRock}(G, 200*\text{milli*darcy}, 0.2);
\end{align*}
\]

Because MRST works in SI units, we must convert from the field units 'darcy' to the SI unit 'meters²'. Alternative: use the conversion function convertFrom(200, milli*darcy)

Homogeneous, anisotropic permeability can be specified in the same way:

\[
\begin{align*}
\text{rock} &= \text{makeRock}(G, [100 100 10].*\text{milli*darcy}, .2);
\end{align*}
\]

Warning: It is better to use makeRock instead of setting rock.poro and rock.perm directly to avoid unintentionally copying data elements from existing rock objects.
Example: heterogeneous model

Generate $\phi$ as a Gaussian field and then compute $K$ from the Carman–Kozeny relation

$$K = \frac{1}{72\tau} \frac{\phi^3 d_p^2}{(1 - \phi)^2},$$

In MRST:

```plaintext
G = cartGrid([50 20]);
p = gaussianField(G.cartDims, ...[0.2 0.4], [11 3], 2.5);
K = p.^3.*(1e-5)^2./(0.81*72*(1-p).^2);
rock = makeRock(G, K(:), p (:));
```

MRST only has very simplified support for geostatistics. For more realistic geostatistics, you should consider commercial software or e.g., GSLIB

```plaintext
plotCellData(G,rock.poro);
colorbar('horiz'); axis equal tight;
plotCellData(G,convertTo(rock.perm,milli*darcy));
colorbar('horiz'); axis equal tight; view(3);
```
Generate a layered model with a single fault in the middle.

\[
G = \text{processGRDECL}(\text{simpleGrdecl}([50, 30, 10], 0.12));
\]
\[
K = \text{logNormLayers}(G.\text{cartDims}, [100, 400, 50, 350], 'indices', [1, 2, 5, 7, 11]);
\]

Four layers with mean values: 100, 400, 50, and 350 mD (top to bottom), and layer thickness: one, three, two, and four grid cells.
Example: Model 2, 10th SPE Comparative Solution Project

Separate module, spe10, for downloading and accessing this model

```plaintext
mrstModule add spe10; rock = SPE10_rock();
```

![Diagram of lateral and vertical permeability](image)
Example: model from Eclipse input deck

Download the SAIGUP data set using `mrstDatasetGUI`. List of files:

```
028_A11.EDITNNC  028.MULTX  028.PERMX  028.SATNUM  SAIGUP.GRDECL
028_A11.EDITNNC.001  028.MULTY  028.PERMY  SAIGUP_A1.ZCORN
028_A11.TRANX  028.MULTZ  028.PERMZ  SAIGUP.ACTNUM
028_A11.TRANY  028.NTG  028.PORO  SAIGUP.COORD
```

Use `deckformat` module to read data

```matlab
mrstModule add deckformat;
grdecl = readGRDECL(fullfile(getDatasetPath('SAIGUP'),'SAIGUP.GRDECL'))
```

```
grdecl =
cartDims: [40 120 20]
COORD: [29766x1 double]
ZCORN: [768000x1 double]
ACTNUM: [96000x1 int32]
PERMX: [96000x1 double]
PERMY: [96000x1 double]
PERMZ: [96000x1 double]
MULTX: [96000x1 double]
MULTY: [96000x1 double]
MULTZ: [96000x1 double]
PORO: [96000x1 double]
NTG: [96000x1 double]
SATNUM: [96000x1 double]
```
Example: synthetic shallow-marine model

The SAIGUP model uses the Eclipse 'METRIC' conventions (permeability in unit md, etc), so we first convert to SI units

```python
usys = getUnitSystem('METRIC');
grdecl = convertInputUnits(grdecl, usys);
```

Then we generate a space-filling grid and extract petrophysical properties

```python
G = processGRDECL(grdecl);
G = computeGeometry(G);
rock = grdecl2Rock(grdecl, G.cells.indexMap);
```
Example: synthetic shallow-marine model

- **Horizontal permeability**
- **Vertical permeability**
- **Net-to-gross**
- **Vertical multipliers less than unity**
Grids in MRST: fully unstructured

All grids are assumed to be unstructured. Basic representation:

- Choices in grid representation are guided by utility and convenience in low-order finite-volume methods
- Available geometric information: centroids, normals, areas, and volumes
- Heavy use of indirection maps
- Redundant information must be constructed, e.g., using run-length encoding
The cell structure, G.cells, has the mandatory fields:

- **num** – the number $N_c$ of cells in the global grid

- **facePos** – indirection map into the faces array. Information of cell $i$ is found in submatrix `faces(facePos(i):facePos(i+1)-1,:)`

  The number of faces of each cell may be computed using the statement `diff(facePos)`. Likewise, the total number of faces is given as $n_f = facePos(end)-1$

- **faces** – $n_f \times 3$ array of global faces connected to a given cell. Specifically, if `faces(i,1)==j`, then global face number `faces(i,2)` is connected to global cell number $j$.

  The third column is optional and can for certain types of grids contain a tag used to distinguish face directions.

  The first column is redundant: cell index $j$ is simply repeated $\text{facePos}(j+1) - \text{facePos}(j)$ times. To conserve memory, we regenerate it using run-length encoding:
  
  ```matlab
  rldecode(1:G.cells.num, diff(G.cells.facePos),2).
  ```
Grid structure: cells

Optional field:

- `indexMap` – \( N_c \times 1 \) array mapping internal cell indices to external cell indices. For models with no inactive cells, `indexMap` equals \( 1 : N_c \). For cases with inactive cells, `indexMap` contains the indices of the active cells sorted in ascending order.

For logically Cartesian grids, a map of cell numbers to logical indices can be constructed using the following statements in 3D:

\[
[ijk\{1:3\}] = \text{ind2sub}(\text{dims}, \text{G.cells.indexMap}(::));
\]
\[
ijk = [ijk\{::\}];
\]

Here, \( ijk(i,:) \) is the global \((I, J, K)\) index of cell \( i \).

Additional fields, typically added by a call to `computeGeometry`:

- `volumes` – an \( N_c \times 1 \) (double) array of cell volumes
- `centroids` – an \( N_c \times d \) (double) array of cell centroids
Grid structure: faces

The face structure, G.faces, has the mandatory fields:

- **num** – the number $N_f \times 1$ of cells in the global grid

- **facePos** – indirection map into the **nodes** array.

- **nodes** – an $N_n \times 2$ array of vertices. If $\text{nodes}(i,1)==j$, local vertex $i$ is part of global face number $j$ and corresponds to global vertex $\text{nodes}(i,2)$. Nodes are oriented such that a right-hand rule determines the direction of the face normal. First column is redundant

- **neighbors** – $N_f \times 2$ array. Global face $i$ is shared by global cells $\text{neighbors}(i,1)$ and $\text{neighbors}(i,2)$. One of the entries in each row can be zero, but not both, to indicate that this is an external face belonging to only one cell (the nonzero entry).
Grid structure: faces and nodes

Optional field:
- tag – can contain user-defined face indicators

Additional fields, typically added by a call to computeGeometry:
- areas – an $N_f \times 1$ of face areas
- normals – an $N_f \times d$ of area weighted, directed face normals, which on face $i$ points from cell neighbors($i,1$) to cell neighbors($i,2$).
- centroids – an $N_f \times d$ array of face centroids.

The vertex structure, G.nodes, consists of two fields:
- num – number $N_n$ of global nodes in the grid
- coords – an $N_n \times d$ array of physical nodal coordinates. Global node $i$ is at physical coordinate coords($i,:)$. 
Example: grid structure

$G = \text{removeCells}(\text{cartGrid}([3,2]), 2)$

$G =$

- cells: [1x1 struct]
- faces: [1x1 struct]
- nodes: [1x1 struct]
- cartDims: [3 2]
- type: {‘tensorGrid’ ‘cartGrid’ ‘removeCells’}
- griddim: 2

\[ \begin{array}{cccccccc}
1 & 1 & 1 & \text{(East)} & 1 & 1 & 0 & 1 \\
1 & 9 & 3 & \text{(South)} & 1 & 5 & 1 & 0 \\
1 & 2 & 2 & \text{(West)} & 2 & 2 & 0 & 2 \\
1 & 11 & 4 & \text{(North)} & 2 & 6 & 2 & 0 \\
2 & 3 & 1 & \text{(East)} & 3 & 3 & 0 & 3 \\
2 & 10 & 3 & \text{(South)} & 3 & 7 & 3 & 4 \\
2 & 4 & 2 & \text{(West)} & 4 & 4 & 4 & 5 \\
2 & 13 & 4 & \text{(North)} & 4 & 8 & 5 & 0 \\
3 & 5 & 1 & \text{(East)} & 5 & 5 & 0 & 1 \\
3 & 11 & 3 & \text{(South)} & 5 & 9 & 0 & 2 \\
3 & 6 & 2 & \text{(West)} & 6 & 6 & 1 & 3 \\
3 & 14 & 4 & \text{(North)} & 6 & 10 & 0 & 4 \\
4 & 6 & 1 & \text{(East)} & 7 & 7 & 2 & 5 \\
4 & 12 & 3 & \text{(South)} & 7 & 11 & 3 & 0 \\
4 & 7 & 2 & \text{(West)} & 8 & 8 & 4 & 0 \\
4 & 15 & 4 & \text{(North)} & 8 & 12 & 5 & 0 \\
5 & 7 & 1 & \text{(East)} & 9 & 2 & : & : \\
5 & 13 & 3 & \text{(South)} & 9 & 1 & : & : \\
5 & 8 & 2 & \text{(West)} & : & : & : & : \\
5 & 16 & 4 & \text{(North)} & : & : & : & : \\
\end{array} \]
Geometry computation: basic steps

Single cell

Tessellation of faces

Area-weighted centroid and normal vector

Triangulation of cell volume
Computer exercises

- List all tutorials in mrst-core and go through at least one of each of the following types:

  datasets/show<name>.m grids/gridTutorial<name>.m

- Make the grid below. Hint: the grid spacing in the $x$-direction is given by $\Delta x (1 - \frac{1}{2} \cos(\pi x))$ and the colors signify cell volumes.

![Grid Example](image)

- Create MRST grids from the standard data set trimesh2d. How would you assign lognormal petrophysical parameters to these grids so that the spatial correlation is preserved?
Outline

1 Introduction
2 Getting started with MRST
3 Grids and petrophysical data
4 Incompressible flow
5 Multiphase flow
6 Compressible flow
7 The AD-OO framework in MRST
What you will learn in this section

We will go through:

- discrete differential and averaging operators
- finite volume methods for $-\nabla (K \nabla p) = q$
- automatic differentiation
- flow solvers in the *incomp* family

You will also get a tast of efficient vectorization tricks in MATLAB

To learn more:
- watch the videos in Jolt1
- study the 1ph tutorials/examples in the *incomp* module
- read Lie et al. (COMG, 2012), doi: 10.1007/s10596-011-9244-4
- read Chapters 4 to 6 in the MRST book
Starting point: mapping $F$ from cell to faces, and $C$ from face to cells:

\[
C = G\text{.faces.neighbors}; \quad \% \text{Cells belonging to each face}
\]
\[
C = C(\text{all}(C \sim= 0, 2), :); \quad \% \text{Only interior faces}
\]
\[
cn = \text{gridCellNo}(G); \quad \% \text{Repeat cell number for all faces}
\]
\[
F = G\text{.cells.faces}(:,1); \quad \% \text{Faces making up each cell}
\]
\[
[nf,nc] = \text{deal(size}(C,1), G\text{.cells.num}); \quad \% \text{Number of faces/cells}
\]

We only consider internal faces, mapping $F$ is represented as $cn$ and $F$. 

---

**Discrete differentiation operators**

**Grid structure in MRST**

**Idealized models**

**Industry models**
The discrete $\text{div}$ operator is a linear mapping from faces to cells:

$$
\text{div}(v)[c] = \sum_{f \in F(c)} \text{sgn}(f)v[f],
$$

$$
\text{sgn}(f) = \begin{cases} 
1, & \text{if } c = C_1(f), \\
-1, & \text{if } c = C_2(f). 
\end{cases}
$$

Here, $v[f]$ denotes a discrete flux over face $f$ with orientation from cell $C_1(f)$ to cell $C_2(f)$. 
The discrete \( \text{grad} \) operator maps from cell pair \( C_1(f), C_2(f) \) to face \( f \):

\[
\text{grad}(p)[f] = p[C_2(f)] - p[C_1(f)],
\]

where \( p[c] \) is a scalar quantity associated with cell \( c \)
The div and grad operators are linear and can be represented as sparse matrix multiplications:

\[
D = \text{sparse}(\text{ones}(\text{nf}, 1) \ast [-1 1], \text{nf}, \text{nc});
\]

\[
\text{grad} = @\text{(x)} D*\text{x};
\]

\[
\text{div} = @\text{(x)} -D'^*\text{x};
\]

With no-flow boundaries, the two operators are adjoint of each other, as in the continuous case.
Finite-volume method: single-phase flow

Fundamental physics: Darcy’s law

\[ \int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = - \int_{\Gamma_f} K(x) \nabla p \cdot \vec{n}_f \, ds \]

\[ \vec{v}[f] = -T[f] \text{grad}(p)[f] \]

Conservation of mass:

\[ \int_{\partial \Omega_c} \vec{v} \cdot \vec{n} \, ds = \int_{\Omega_c} \nabla \cdot \vec{v} \, d\vec{x} = \int_{\Omega_c} q \, d\vec{x} \]

\[ \text{div}(\vec{v})[c] = q[c] \]

\[ T_{i,k} = A_{i,k} \frac{\vec{c}_{i,k} \cdot K_i \vec{n}_{i,k}}{|\vec{c}_{i,k}|^2} \]

\[ T_{i,k} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1} \]
Finite-volume method: single-phase flow

Fundamental physics: Darcy’s law

$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = - \int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

$$\vec{v}[f] = -T[f] \text{grad}(p)[f]$$

Conservation of mass:

$$\int_{\partial \Omega_c} \vec{v} \cdot \vec{n} \, ds = \int_{\Omega_c} \nabla \cdot \vec{v} \, d\bar{x} = \int_{\Omega_c} q \, d\bar{x}$$

$$\text{div}(\vec{v})[c] = q[c]$$

We start by extracting face normals and vectors from cell to face centroids:

```matlab
sgn = 2*(cn == G.faces.neighbors(F, 1)) - 1;
c = G.faces.centroids(F,:) - G.cells.centroids(cn,:);
n = bsxfun(@times, sgn, G.faces.normals(F,:));
```

Here, the first line determines the correct sign of the face normal
Finite-volume method: single-phase flow

Fundamental physics: Darcy’s law

\[
\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = - \int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds
\]

\[
\vec{v}[f] = - \mathbf{T}[f] \cdot \text{grad}(p)[f]
\]

Conservation of mass:

\[
\int_{\partial \Omega_c} \vec{v} \cdot \vec{n} \, ds = \int_{\Omega_c} \nabla \cdot \vec{v} \, d\vec{x} = \int_{\Omega_c} q \, d\vec{x}
\]

\[
\text{div}(\vec{v})[c] = q[c]
\]

We start by extracting face normals and vectors from cell to face centroids:

\[
\text{sgn} = 2 \times (\text{cn} == \text{G.faces.neighbors(F, 1)}) - 1;
\]

\[
c = \text{G.faces.centroids(F,:)} - \text{G.cells.centroids(cn,:)};
\]

\[
n = \text{bsxfun(@times, sgn, G.faces.normals(F,:))};
\]

Here, the first line determines the correct sign of the face normal

**Extract permeability vector** \([K_{xx}, K_{xy}, K_{yx}, K_{yy}]\) for each cell:

\[
[K, i, j] = \text{permTensor(rock, G.griddim)};
\]
Finite-volume method: single-phase flow

Fundamental physics: Darcy’s law

\[
\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = - \int_{\Gamma_f} K(x) \nabla p \cdot \vec{n}_f \, ds
\]

\[\vec{v}[f] = -T[f] \text{grad}(p)[f]\]

Conservation of mass:

\[
\int_{\partial \Omega_c} \vec{v} \cdot \vec{n} \, ds = \int_{\Omega_c} \nabla \cdot \vec{v} \, dx = \int_{\Omega_c} q \, dx
\]

\[\text{div}(\vec{v})[c] = q[c]\]

Then, we can compute the one-sided transmissibilities \(T_{i,k}\):

% In 2D: i=[1 1 2 2], j=[1 2 1 2]
\[
hT = \text{sum}(c(:,i)) .* \text{bsxfun(@times, K(cn,:), n(:,j))), 2); 
\]
\[
hT = hT./ \text{sum}(c.*c,2);
\]

Here, \text{bsxfun} applies an element-by-element multiply operation to two arrays. Compact notation for a double for-loop
Finite-volume method: single-phase flow

Fundamental physics: Darcy’s law

\[ \int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = - \int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds \]

\[ \vec{v}[f] = - \mathbf{T}[f] \text{grad}(p)[f] \]

Conservation of mass:

\[ \int_{\partial \Omega_c} \vec{v} \cdot \vec{n} \, ds = \int_{\Omega_c} \nabla \cdot \vec{v} \, d\vec{x} = \int_{\Omega_c} q \, d\vec{x} \]

\[ \text{div}(\vec{v})[c] = q[c] \]

Then, we can compute the one-sided transmissibilities \( T_{i,k} \):

% In 2D: i=[1 1 2 2], j=[1 2 1 2]
\[ hT = \text{sum}(c(:,i)) .* \text{bsxfun(@times, K(cn,:), n(:,j))), 2); \]
\[ hT = hT./ \text{sum}(c.*c,2); \]

Here, \text{bsxfun} applies an element-by-element multiply operation to two arrays. Compact notation for a double for-loop

One-side transmissibilities can be computed once using the function:

\[ hT = \text{computeTrans}(G, \text{rock}); \]
Finite-volume method: single-phase flow

Fundamental physics: Darcy’s law

\[ \int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = - \int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds \]

\[ \vec{v}[f] = - \mathbf{T}[f] \text{grad}(p)[f] \]

Conservation of mass:

\[ \int_{\partial \Omega_c} \vec{v} \cdot \vec{n} \, ds = \int_{\Omega_c} \nabla \cdot \vec{v} \, dx = \int_{\Omega_c} q \, dx \]

\[ \text{div}(\vec{v})[c] = q[c] \]

Finally, we can compute the transmissibilities \( T_{i,k} \):

\[
T = 1 ./ \text{accumarray}(F, 1 ./ hT, [G.faces.num, 1]);
T = T(all(C==0,2),:);
\]

Here, \( \text{accumarray}(p,v) \) collects all elements of \( v \) that have identical subscripts in \( p \), sums them, and stores the result in the location given by \( p \).
Finite-volume method: single-phase flow

Fundamental physics: Darcy’s law
\[
\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds
\]
\[
\vec{v}[f] = -
\mathbf{T}[f] \text{grad}(p)[f]
\]

Conservation of mass:
\[
\int_{\partial \Omega_c} \vec{v} \cdot \vec{n} \, ds = \int_{\Omega_c} \nabla \cdot \vec{v} \, dx = \int_{\Omega_c} q \, dx
\]
\[
\text{div}(\vec{v})[c] = q[c]
\]

Usually, you would not have to implement all this, but rather call a function that also includes various safeguards:

\[
S = \text{setupOperatorsTPFA}(G, \text{rock});
\]
Discretization of flow models leads to large system of (non)linear equations. Can be linearized and solved with Newton’s method

\[ F(x) = 0 \quad \Rightarrow \quad \frac{\partial F}{\partial x}(x^i)(x^{i+1} - x^i) = -F(x^i) \]

Coding necessary Jacobians is time-consuming and error prone

Automatic differentiation

- Idea: keep track of variables and derivatives simultaneously
- Any code, regardless of complexity, can be broken down to a limited set of arithmetic operations (\(+, -, *, /, \ldots\)) and elementary functions (\(\text{sin}, \exp, \text{power}, \ldots\))
- Derivative rules are known for these operations and functions
- Combine these with the chain rule
Automatic differentiation

- Consider a scalar primary variable $x$ and a function $f(x)$, whose AD representations are the pairs $\langle x, 1 \rangle$ and $\langle f, f_x \rangle$

- Must define the action of elementary operations and functions on all such pairs:

$$
\langle f, f_x \rangle + \langle g, g_x \rangle = \langle f + g, f_x + g_x \rangle,
\langle f, f_x \rangle \langle g, g_x \rangle = \langle fg, fg_x + f_x g \rangle,
\exp \langle \langle f, f_x \rangle \rangle = \langle \exp \langle f \rangle, \exp \langle f \rangle f_x \rangle,
\sin \langle \langle f, f_x \rangle \rangle = \langle \sin \langle f \rangle, \cos \langle f \rangle f_x \rangle,
$$

- Use operator overloading to write $a + b \times c$ rather than awkward constructs like `myPlus(a,myTimes(b,c))`

- Goal: make code as close as possible to mathematical description of model
Implementation of AD in MRST

\[ [x, y] = \text{initVariablesADI}(1, 2); \]
\[ z = 3 \times \exp(-x \times y) \]

- \( x \) = ADI Properties:
  - val: 1
  - jac: \{[1] [0]\}

- \( y \) = ADI Properties:
  - val: 2
  - jac: \{[0] [1]\}

- \( z \) = ADI Properties:
  - val: 0.4060
  - jac: \{[-0.8120] [-0.4060]\}

MRST implementation tailored to reservoir simulation and MATLAB:
- designed to be efficient for vector variables more than scalars
- works with sub-Jacobians rather than full Jacobians to simplify subsequent manipulation
Applying AD to incompressible flow

First, we write the flow equation as a residual:

\[ F(p) = \nabla \cdot (K \nabla p) + q = 0 \]

In discrete form:

\[ F(p) = \text{div}(T \text{grad}(p)) + q = Ap + q = 0 \]

Apply Newton’s method with a zero initial guess:

\[ \frac{\partial F}{\partial p}(p)(p - 0) = -F(0) \quad \Leftrightarrow \quad Ap = -q \]

Using AD means that we never need to form \( A \) explicitly
Solving the Poisson equation: $-\Delta p = q$

% Grid and grid information
G = cartGrid([5 5]);
G = computeGeometry(G);
rock = makeRock(G, 1, 1);
nc = G.cells.num;

% Operators
S = setupOperatorsTPFA(G, rock);
spy(S.C);

% Assemble and solve equations
p = initVariablesADI(zeros(nc, 1));
q = zeros(nc, 1); % source term
q([1 nc]) = [1 -1]; % quarter five-spot

eq = S.Div(S.Grad(p)) + q; % equation
eq(1) = eq(1) + p(1); % make solution unique
p = -eq.jac{1}\eq.val; % solve equation
plotCellData(G, p);
Solving the Poisson equation: non-rectangular domain

% Grid and grid information
G = computeGeometry(cartGrid([20 20], [1 1]));
r1 = sum(bsxfun(@minus, G.cells.centroids,[0 .5 1]).^2,2);
r2 = sum(bsxfun(@minus, G.cells.centroids,[0 .5 0]).^2,2);
G = extractSubgrid(G, (r1>0.16) & (r2>0.16));
rock = makeRock(G, 1, 1);
nc = G.cells.num;

% Operators
S = setupOperatorsTPFA(G,rock);
spy(S.C);

% Assemble and solve equations
p = initVariablesADI(zeros(nc,1));
q = zeros(nc, 1); % source term
q([1 nc]) = [1 -1]; % quarter five-spot

eq = S.Div(S.Grad(p))+q; % equation
eq(1) = eq(1) + p(1); % make solution unique
p = -eq.jac{1}\eq.val; % solve equation
plotCellData(G,p);
Solving the Poisson equation: unstructured grid

% Grid and grid information
load seamount
G = pebi(triangleGrid([x(:) y(:)]));
G = computeGeometry(G);
rock = makeRock(G, 1, 1);
nc = G.cells.num;

% Operators
S = setupOperatorsTPFA(G, rock);
spy(S.C);

% Assemble and solve equations
p = initVariablesADI(zeros(nc, 1));
q = zeros(nc, 1)
q([135 282 17]) = [-1 0.5 0.5];

eq = S.Div(S.T.*S.Grad(p))+q;
eq(1) = eq(1) + p(1);
p = -eq.jac{1}\eq.val;
plotCellData(G,p);
Switching between different discretization schemes

Discretization schemes: represented in terms of discrete $\text{div}$ and $\text{grad}$ operators, and some discrete representation of a bilinear form, e.g., $(\vec{u}, \vec{v}) = \int \vec{u} \cdot K^{-1} \vec{v}$.

As example, consider: \[ \nabla \cdot \vec{u} = q, \quad \vec{u} = -K \nabla p \]

**Two-point flux approximation (TPFA)**

Given a vector $T_{tp}$ of transmissibilities, the coded equations become

\[
\begin{align*}
v & = -T_{tp} \cdot \text{grad}(p) \\
eq & = \text{div}(v) - q;
\end{align*}
\]

\[
\begin{align*}
eq & = \text{div}(T_{tp} \cdot \text{grad}(p)) + q;
\end{align*}
\]

**Multi-point flux approximation (MPFA)**

Given a matrix $T_{mp}$ of transmissibilities, the coded equations become

\[
\begin{align*}
v & = -T_{mp} \cdot \text{grad}(p) \\
eq & = \text{div}(v) - q;
\end{align*}
\]

\[
\begin{align*}
eq & = \text{div}(T_{mp} \cdot \text{grad}(p)) + q;
\end{align*}
\]
Switching between different discretization schemes, cont’d

Same example: \[ \nabla \cdot \vec{u} = q, \quad \vec{u} = -K \nabla p \]

**Lowest order mixed (or mimetic) formulation**

Given a matrix \( M \) with \( m_{ij} \approx \int \vec{\psi}_i \cdot K^{-1} \vec{\psi}_j \), one may be tempted to code the equations as

\[
\begin{align*}
  v &= -M \\text{grad}(p); \\
  \text{eq} &= \text{div}(v) - q;
\end{align*}
\]

... will work but involves applying \( M^{-1} \) to the \( n_f \times n_c \) grad-matrix. Instead let flux \( v \) be primary variable, and solve for both \( v \) and \( p \):

\[
\begin{align*}
  [v, p] &= \text{initVariablesADI}(\text{zeros}(nf, 1), \text{zeros}(nc, 1)); \\
  \text{eq}{1} &= M*v + \text{grad}(p); \quad \% \text{ Darcy's law} \\
  \text{eq}{2} &= \text{div}(v) - q; \quad \% \text{ continuity equation} \\
  \text{eq} &= \text{cat} \{{eq\{1\}, ...}\}; \quad \% \text{ concatenate equations} \\
  x &= -\text{eq}.\text{jac}{1}\text{\textbackslash eq.val}; \quad \% \text{ solve, } x \text{ contains both } v \text{ and } p
\end{align*}
\]
More advanced problems

You can continue to expand these examples with more effects:
- buoyancy effects and fluid viscosity
- source terms and boundary conditions
- well models
- multiphase mobilities
- transmissibility multipliers
- ...

Eventually, the code will become quite involved and you will need to encapsulate your implementation inside functions (or objects)

This is done in the \texttt{incomp} family of solvers:

\texttt{incompTPFA, incompMimetic, incompMPFA,...}
Assembly of linear equation in *incomp* solvers

Automatic differentiation and discrete operators are new ideas in MRST. The *incomp* solver family uses mechanical assembly. For TPFA:

- We sum the transmissibilities of all faces to create the diagonal:

  \[
  d = \text{accumarray}([C(:,1); C(:,2)], \text{repmat}(T,[2,1]),[nc, 1]);
  \]

- Then, we construct the discrete matrix

  \[
  I = [C(:,1); C(:,2); (1:nc)'];
  J = [C(:,2); C(:,1); (1:nc)'];
  V = [-T; -T; d]; \text{clear } d;
  A = \text{sparse(double(I), double(J), V, nc, nc)};
  \]

- Assuming we know the right-hand side, we can solve the flow equation:

  \[
  A(1) = 2*A(1); \quad \% \text{Set } p=0 \text{ in cell #1 if only Dirichlet b.c}
  p = \text{mldivide}(A, \text{rhs});
  \]
Basic data structures in simulation models

- **Fluid properties:**

  ```
  fluid = initSingleFluid('mu', 1*centi*poise, ...
  'rho', 1014*kilogram/meter^3);
  ```

- **Reservoir states (physical variables):**

  ```
  state = initResSol(G, p0, s0);
  state = initState(G, W, p0, s0);
  ```

- **Fluid sources**

  ```
  src = addSource(src, cells, rates);
  src = addSource(src, cells, rates, 'sat', sat);
  ```
Basic data structures in simulation models

- **Boundary conditions**

```plaintext
bc = addBC(bc, faces, type, values);
bc = addBC(bc, faces, type, values, 'sat', sat);
```

For grids having logical $IJK$ numbering:

```plaintext
bc = pside(bc, G, side, p);
bc = fluxside(bc, G, side, flux)
```

where side would be 1/’West’/’XMin’/’Left’, etc

- **Wells with Peacemmann well model:**

```plaintext
W = addWell(W, G, rock, cellInx);
W = addWell(W, G, rock, cellInx, 'pn', pv, ...);
```

For convenience, we also have the functions

```plaintext
W = verticalWell(W, G, rock, I, J, K)
W = verticalWell(W, G, rock, I, K)
```
The three main solvers available are:

- **The standard two-point solver:**

  ```plaintext
  mrstModule add incomp;
  hT = computeTrans(G, rock);
  state = incompTPFA(state, G, hT, fluid, ...);
  ```

- **Lowest-order mimetic finite-difference methods:**

  ```plaintext
  mrstModule add mimetic;
  IP = computeMimeticIP(G, rock);
  state = incompMimetic(state, G, IP, fluid, ...);
  ```

- **The MPFA-O multipoint flux-approximation method:**

  ```plaintext
  mrstModule add mpfa;
  hT = computeMultiPointTrans(G, rock);
  state = incompMPFA(state, G, hT, fluid)
  ```
Example: quarter five-spot with source terms

```matlab
gravity reset off
[nx,ny] = deal(20);
G = cartGrid([nx,ny],[500,500]);
G = computeGeometry(G);
rock = makeRock(G, 100*milli*darcy, .2);
hT = computeTrans(G, rock);

fluid = initSingleFluid('mu', 1*centi*poise, ...
    'rho', 1014*kilogram/meter^3);
pv = sum(poreVolume(G,rock));
src = addSource([], 1, pv);
src = addSource(src, G.cells.num, -pv);

state = initResSol(G, 0.0, 1.0);
state = incompTPFA(state, G, hT, fluid, 'src', src);
plotCellData(G, state.pressure);
plotGrid(G, src.cell, 'FaceColor', 'w');

mrstModule add streamlines;
seed = (nx:nx-1:nx*ny).';
Sf = pollock(G, state, seed, 'substeps', 1);
Sb = pollock(G, state, seed, 'substeps', 1, 'reverse', true);
h=streamline([Sf; Sb]); set(h, 'Color', 'k');
```
Example: horizontal and vertical well

\[
[nx, ny, nz] = \text{deal}(20, 20, 5);
G = \text{cartGrid}([nx, ny, nz], [500 500 25])
G = \text{computeGeometry}( ) ;
:\hT = \text{computeTrans}(G, \text{rock});
\]
\[
W = \text{verticalWell}([], G, \text{rock}, 1, 1, 1:nz, ... \\
\quad \text{'Type'}, \text{'rate'}, \text{'Comp_i'}, 1, ... \\
\quad \text{'Val'}, 3e3/\text{day}, ... \\
\quad \text{'Radius'}, .12*\text{meter}, \text{'name'}, [1] );
\]
\[
W = \text{addWell}(W, G, \text{rock}, nx : ny : nx*ny, ... \\
\quad \text{'Type'}, \text{'bhp'}, \text{'Comp_i'}, 1, ... \\
\quad \text{'Val'}, 1.0e5, \text{'Radius'}, .12*\text{meter}, ... \\
\quad \text{'Dir'}, \text{'y'}, \text{'name'}, \text{'P'});
\]

gravity reset on;
state = \text{initState}(G, W, 0);
state = \text{incompTPFA}(state, G, \hT, \text{fluid}, [\text{wells}], W);

\[
\text{plotCellData}(G, \text{state}.pressure, ['EdgeAlpha', .01, 'FaceAlpha', .4]);
\]
\[
\text{plotWell}(G, W(1), ['radius', 1, 'color', 'r']);
\]
\[
\text{plotWell}(G, W(2), ['radius', .5, 'color', 'b']);
\]
\[
\text{view}(3), \text{camproj\ perspective, axis\ tight\ off}
\]
Run the quarter five-spot example with the following modifications:

1. Replace the Cartesian grid by a curvilinear grid, e.g., use twister or a random perturbation of internal nodes as shown in the lectures. Do you see differences if you replace TPFA with mimetic of MPFA?

2. Set the domain to be a single layer of the SPE 10 model. Hint: use getSPE10rock() to sample the petrophysical parameters.

Notice that pollock may not work for non-Cartesian grids and you may wish to compute time-of-flight instead using the diagnostics module.

Pick a bed model from BedModels1 or BedModel2. Compute flow subject to linear pressure drop first in the $x$ and then in the $y$-direction. A unit pressure drop is the most wide-spread computational setup used for flow-based upscaling.

Explore the many tutorial examples found in mrst-core, incomp and mrst-book/1phase
1. Introduction
2. Getting started with MRST
3. Grids and petrophysical data
4. Incompressible flow
5. Multiphase flow
6. Compressible flow
7. The AD-OO framework in MRST
What you will learn in this section

In this section, we study incompressible, two-phase flow

You will learn about:

- discretizing the transport equation on unstructured grids
- nonlinear solution strategy (Newton–Raphson, time-step control)
- implementation in MRST

To learn more:
- study the 2ph tutorials/examples in the incomp module
- read Chapters 8 to 10 in the MRST book
Two-phase, incompressible flow

The solvers of the incomp family are designed to solve two-phase models consisting of an elliptic pressure equation

$$\nabla \cdot \vec{v} = q, \quad \vec{v} = -\lambda (\nabla p_n - f_w \nabla P_c - (\rho_w f_w + \rho_n f_n) g \nabla z)$$

and a hyperbolic/parabolic transport equation

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \left[ f_w (\vec{v} + \lambda_n (\Delta \rho g \nabla z + \nabla P_c)) \right] = q_w$$
Two-phase, incompressible flow

The solvers of the incomp family are designed to solve two-phase models consisting of an elliptic pressure equation

$$\nabla \cdot \vec{v} = q, \quad \vec{v} = -\lambda (\nabla p_n - f_w \nabla P_c - (\rho_w f_w + \rho_n f_n) g \nabla z)$$

and a hyperbolic/parabolic transport equation

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \left[ f_w (\vec{v} + \lambda_n (\Delta \rho g \nabla z + \nabla P_c)) \right] = q_w$$

Standard approach – sequential solution procedure:

<table>
<thead>
<tr>
<th>Compute initial state and set $t = 0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>While $t &lt; T$</td>
</tr>
<tr>
<td>Fix $S_w$ and solve elliptic pressure equation</td>
</tr>
<tr>
<td>Fix $p$ and $\vec{v}$ and solve transport equation a time $\Delta t$</td>
</tr>
<tr>
<td>$t = t + \Delta t$</td>
</tr>
</tbody>
</table>
Two-phase, incompressible flow

The solvers of the incomp family are designed to solve two-phase models consisting of an elliptic pressure equation

$$\nabla \cdot \vec{v} = q, \quad \vec{v} = -\lambda(\nabla p_n - f_w \nabla P_c - (\rho_w f_w + \rho_n f_n) g \nabla z)$$

and a hyperbolic/parabolic transport equation

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \left[ f_w (\vec{v} + \lambda_n (\Delta \rho g \nabla z + \nabla P_c)) \right] = q_w$$

For the pressure equation, we use same methods as discussed above with obvious modifications. Solvers implemented for multiphase elliptic pressure equation:

incompTPFA, incompMimetic, incompMPFA,...

The only changes are in how mobility and right-hand side are computed
Two-phase, incompressible flow

The solvers of the \texttt{incomp} family are designed to solve two-phase models consisting of an elliptic pressure equation

\[
\nabla \cdot \vec{v} = q, \quad \vec{v} = -\lambda \left( \nabla p_n - f_w \nabla P_c - (\rho_w f_w + \rho_n f_n) g \nabla z \right)
\]

and a hyperbolic/parabolic transport equation

\[
\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \left[ f_w \left( \vec{v} + \lambda_n (\Delta \rho g \nabla z + \nabla P_c) \right) \right] = q_w
\]

For the transport equation, we have two different solvers:

\[
\begin{align*}
\text{state} &= \text{explicitTransport}(\text{state}, \text{G}, \text{tf}, \text{rock}, \text{fluid}, \text{\'mech1\'}, \text{obj1}, \ldots) \\
\text{state} &= \text{implicitTransport}(\text{state}, \text{G}, \text{tf}, \text{rock}, \text{fluid}, \text{\'mech1\'}, \text{obj1}, \ldots)
\end{align*}
\]

designed to work on fully unstructured grids, but only implemented for two-phase flow
Discretization of $\phi S_t + \nabla \cdot \vec{H}(S) = 0$

Flux $\vec{H}$ incorporates effects of viscous, gravity, and capillary forces:

$$\vec{H}(S) = f(s)\vec{v} + \frac{\lambda_w \lambda_n K}{\lambda_w + \lambda_n} (\Delta \rho \vec{g} + \nabla P_c(S))$$

$$= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).$$
Discretization of \( \phi S_t + \nabla \cdot \vec{H}(S) = 0 \)

Flux \( \vec{H} \) incorporates effects of viscous, gravity, and capillary forces:

\[
\vec{H}(S) = f(s)\vec{v} + \frac{\lambda_w \lambda_n K}{\lambda_w + \lambda_n} (\Delta \rho \vec{g} + \nabla P_c(S)) \\
= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).
\]

Integrated over cell \( \Omega_i \) and in time

\[
S_{i}^{n+1} - S_{i}^{n} = \frac{1}{\phi_i |\Omega_i|} \sum_k \int_{t_n}^{t_{n+1}} \int_{\Gamma_{i,k}} \vec{H}(S(\vec{x}, t)) \cdot \vec{n}_{i,k} \, ds \, dt
\]
Discretization of $\phi S_t + \nabla \cdot \vec{H}(S) = 0$

Flux $\vec{H}$ incorporates effects of viscous, gravity, and capillary forces:

$$\vec{H}(S) = f(s)\vec{v} + \frac{\lambda_w \lambda_n K}{\lambda_w + \lambda_n} (\Delta \rho \vec{g} + \nabla P_c(S))$$

$$= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).$$

Integrated over cell $\Omega_i$ and in time

$$S_i^{n+1} - S_i^n = \frac{1}{\phi_i |\Omega_i|} \sum_k \int_{t_n}^{t_{n+1}} \int_{\Gamma_{ik}} \vec{H}(S(\vec{x}, t)) \cdot \vec{n}_{i,k} \, ds \, dt$$

For first-order methods, we can evaluate integral at end-points:

$$S_i^{n+1} - S_i^n = \frac{\Delta t}{\phi_i |\Omega_i|} \int_{\Gamma_{ik}} \vec{H}(S(\vec{x}, t_m)) \cdot \vec{n}_{i,k} \, ds, \quad m = n, n + 1$$
Discretization of $\phi S_t + \nabla \cdot \vec{H}(S) = 0$

Flux $\vec{H}$ incorporates effects of viscous, gravity, and capillary forces:

$$\vec{H}(S) = f(s)\vec{v} + \frac{\lambda_w \lambda_n K}{\lambda_w + \lambda_n} (\Delta \rho \vec{g} + \nabla P_c(S'))$$

$$= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).$$

For first-order methods, we can evaluate integral at end-points:

$$S^{m+1}_i - S^m_i = \frac{\Delta t}{\phi_i |\Omega_i|} \int_{\Gamma_{ik}} \vec{H}(S(x, t_m)) \cdot \vec{n}_{i,k} \, ds, \quad m = n, n + 1$$

Capillary term is discretized using the TPFA method:

$$A_{i,k} \nabla P_c(S) \cdot \vec{n}_{i,k} \approx \left[ T_{i,k}^{-1} + T_{k,i}^{-1} \right]^{-1} (P_c(S_i) - P_c(S_k)) = P_{i,k}(S)$$
Discretization of $\phi S_t + \nabla \cdot \mathbf{H}(S) = 0$

Flux $\mathbf{H}$ incorporates effects of viscous, gravity, and capillary forces:

$$\mathbf{H}(S) = f(s)\mathbf{v} + \frac{\lambda_w \lambda_n K}{\lambda_w + \lambda_n} (\Delta \rho \mathbf{g} + \nabla P_c(S))$$

$$= \mathbf{H}_f(S) + \mathbf{H}_g(S) + \mathbf{H}_c(S).$$

For first-order methods, we can evaluate integral at end-points:

$$S_i^{m+1} - S_i^m = \frac{\Delta t}{\phi_i|\Omega_i|} \int_{\Gamma_{i,k}} \mathbf{H}(S(x,t_m)) \cdot \mathbf{n}_{i,k} \, ds,$$

$$m = n, n + 1$$

We define a “gravity flux” $g_{ik}$ that is independent of saturation:

$$g_{ik} = \left[ g_{i,k}^{-1} + g_{k,i}^{-1} \right]^{-1}, \quad g_{i,k} = (\Delta \rho)|_{\Omega_i} K_i \mathbf{g} \cdot \mathbf{n}_{i,k},$$

$$g_{k,i} = (\Delta \rho)|_{\Omega_k} K_k \mathbf{g} \cdot \mathbf{n}_{k,i}.$$
Discretization of \( \phi S_t + \nabla \cdot \vec{H}(S) = 0 \)

Flux \( \vec{H} \) incorporates effects of viscous, gravity, and capillary forces:

\[
\vec{H}(S) = f(s)\vec{v} + \frac{\lambda_w \lambda_n \mathbf{K}}{\lambda_w + \lambda_n} (\Delta \rho \vec{g} + \nabla P_c(S)) \\
= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).
\]

Summing up, we have:

\[
H_{ik} = \frac{\lambda^u_w}{\lambda^u_w + \lambda^u_n} v_{ik} + \frac{\lambda^u_w \lambda^u_n}{\lambda^u_w + \lambda^u_n} \left[ g_{ik} + P_{ik} \right]
\]

where \( \lambda^u_w \) and \( \lambda^u_n \) are upstream-evaluated phase mobilities
Discretization of $\phi S_t + \nabla \cdot \vec{H}(S) = 0$

Flux $\vec{H}$ incorporates effects of viscous, gravity, and capillary forces:

$$\vec{H}(S) = f(s)\vec{v} + \frac{\lambda_w\lambda_n}{\lambda_w + \lambda_n} \left( \Delta \rho \vec{g} + \nabla P_c(S) \right)$$

$$= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).$$

Summing up, we have:

$$H_{ik} = \frac{\lambda^u_w}{\lambda^u_w + \lambda^u_n} v_{ik} + \frac{\lambda^u_w\lambda^u_n}{\lambda^u_w + \lambda^u_n} \left[ g_{ik} + P_{ik} \right]$$

where $\lambda^u_w$ and $\lambda^u_n$ are upstream-evaluated phase mobilities

If sign of $v_{ik}$ and $g_{ik} + P_{ik}$ is different, choose mobilities from opposite sides. Otherwise, check sign of $v_{ik} + \lambda_\alpha (g_{ik} + P_{ik})$ for $\alpha = w, n$
Implementation in MRST

Explicit scheme: $S^{n+1} = S^n - F(S^n, S^n)$. Implicit scheme: $F(S^{n+1}, S^n) = 0$

$$F_i(s, r) = s_i - r_i + \frac{\Delta t}{\phi_i|\Omega_i|} \left[ \sum_k H_{ik}(s) - \max(q_i, 0) - \min(q_i, 0) f(S_i) \right]$$

$$H_{ik}(s) = \frac{\lambda^u(s_i, s_k)}{\lambda^u_w(s_i, s_k) + \lambda^u_n(s_i, s_k)} \left[ v_{ik} + \lambda^u_n(s_i, s_k)(g_{ik} + P_{ik}) \right]$$
Implementation in MRST

Explicit scheme: \( S^{n+1} = S^n - \mathcal{F}(S^n, S^n) \). Implicit scheme: \( \mathcal{F}(S^{n+1}, S^n) = 0 \)

\[
\mathcal{F}_i(s, r) = s_i - r_i + \frac{\Delta t}{\phi_i|\Omega_i|} \left[ \sum_k H_{ik}(s) - \max(q_i, 0) - \min(q_i, 0) f(S_i) \right]
\]

\[
H_{ik}(s) = \frac{\lambda_u(s_i, s_k)}{\lambda_w(s_i, s_k) + \lambda_n(s_i, s_k)} \left[ v_{ik} + \lambda_u(s_i, s_k)(g_{ik} + P_{ik}) \right]
\]

To avoid code duplication, the residual form \( \mathcal{F} \) and its Jacobian \( J = d\mathcal{F} \) are computed in a private helper function:

\[
[F, Jac] = \text{twophaseJacobian}(G, \text{state}, \text{rock}, \text{fluid}, 'pn1', pv1, \ldots)
\]

Code is quite complex since Jacobian is computed explicitly (this was developed before AD was introduced in MRST)
Implementation in MRST

Explicit scheme: \( S^{n+1} = S^n - \mathcal{F}(S^n, S^n) \). Implicit scheme: \( \mathcal{F}(S^{n+1}, S^n) = 0 \)

\[
\mathcal{F}_i(s, r) = s_i - r_i + \frac{\Delta t}{\phi_i|\Omega_i|} \left[ \sum_k H_{ik}(s) - \max(q_i, 0) - \min(q_i, 0)f(S_i) \right]
\]

\[
H_{ik}(s) = \frac{\lambda^u_w(s_i, s_k)}{\lambda^u_w(s_i, s_k) + \lambda^u_n(s_i, s_k)} \left[ v_{ik} + \lambda^u_n(s_i, s_k)(g_{ik} + P_{ik}) \right]
\]

Explicit transport solver:

```matlab
F = twophaseJacobian(G, state, rock, fluid, 'wells', opt.wells, ...);
s = state.s(:,1);
t = 0;
while t < tf,
    dt = min(tf-t, getdt(state));
    s(:) = s - F(state, state, dt);
    t = t + dt;
    s = correct_saturations(s, opt.satwarn);
    state.s = [s, 1-s];
end
```
Explicit scheme: \( S^{n+1} = S^n - F(S^n, S^n) \). Implicit scheme: \( F(S^{n+1}, S^n) = 0 \)

\[
F_i(s, r) = s_i - r_i + \frac{\Delta t}{\phi_i|\Omega_i|} \left[ \sum_k H_{ik}(s) - \max(q_i, 0) - \min(q_i, 0)f(S_i) \right]
\]

\[
H_{ik}(s) = \frac{\lambda_u(s_i, s_k)}{\lambda_w(s_i, s_k) + \lambda_n(s_i, s_k)} \left[ v_{ik} + \lambda_u(s_i, s_k)(g_{ik} + P_{ik}) \right]
\]

Implicit solver uses a Newton method:

\[
0 = F(s_0 + \delta s) \approx F(s_0) + J(s_0)\delta s,
\]

\[
J(s^{\ell})\delta s^{\ell+1} = -F(s^{\ell}), \quad s^{\ell+1} \leftarrow s^{\ell} + \delta s^{\ell+1}
\]
Implementation in MRST

Explicit scheme: \( S^{n+1} = S^n - F(S^n, S^n) \). Implicit scheme: \( F(S^{n+1}, S^n) = 0 \)

\[
F_i(s, r) = s_i - r_i + \frac{\Delta t}{\phi_i|\Omega_i|} \left[ \sum_k H_{ik}(s) - \max(q_i, 0) - \min(q_i, 0)f(S_i) \right]
\]

\[
H_{ik}(s) = \frac{\lambda_u(s_i, s_k)}{\lambda_w(s_i, s_k) + \lambda_u(s_i, s_k)} \left[ v_{ik} + \lambda_u(s_i, s_k)(g_{ik} + P_{ik}) \right]
\]

Implicit solver uses a Newton method:

\[
0 = F(s_0 + \delta s) \approx F(s_0) + J(s_0)\delta s,
\]

\[
J(s^\ell)\delta s^{\ell+1} = -F(s^\ell), \quad s^{\ell+1} \leftarrow s^\ell + \delta s^{\ell+1}
\]

To get saturation values in \([0, 1]\), we need to introduce a line-search method that uses \( p^\ell = \delta s^{\ell+1} \) as search direction. MRST uses an inexact method that asks for a sufficient decrease in \( F(s^\ell + \alpha p^\ell) \) and reduces \( \alpha \) in a geometric sequence.
mints = pow2(tf, -opt.tsref);
[t, dt] = deal(0.0, tf);
while t < tf && dt >= mints,
    dt = min(dt, tf - t);
    redo_newton = true;
    while redo_newton,
        sn_0 = resSol; sn = resSol; sn.s(:) = min(1, sn.s + 0.05);
        res = F(sn, sn_0, dt);
        err = norm(res(:), inf);
        [nwtfail, linfail, it] = deal(err > opt.nltol, false, 0);
        while nwtfail && ~linfail && it < opt.maxnewt,
            J = Jac(sn, sn_0, dt);
            ds = -reshape(opt.LinSolve(J, reshape(res', [], 1)), ns, []);
            [sn, res, alph, linfail] = update(sn, sn_0, ds, dt, err);
            it = it + 1;
            err = norm(res(:), inf);
            nwtfail = err > opt.nltol;
        end
        if nwtfail,
            % Chop time step in two, or use previous successful dt
        else
            redo_newton = false;
            t = t + dt;
            % If five successful steps, increase dt by 50%
        end
    end
end
resSol = sn;
Example: Buckley–Leverett displacement

```matlab
G = computeGeometry(cartGrid([100,1]));
rock = makeRock(G, 100*milli*darcy, 0.2);
fluid = initSimpleFluid('mu', [1, 1].*centi*poise, ...
    'rho', [1000, 1000].*kilogram/meter^3, 'n', [2,2]);
bc  = fluxside([], G, 'Left', 1, 'sat', [1 0]);
bc  = fluxside(bc, G, 'Right', -1, 'sat', [0 1]);
hT  = computeTrans(G, rock);
rSol = initState(G, [], 0, [0 1]);
rSol = incompTPFA(rSol, G, hT, fluid, 'bc', bc);
rSole = explicitTransport(rSol, G, 10, rock, fluid, 'bc', bc, 'verbose',true);
```

![Graph showing Buckley-Leverett displacement](image)
Example: inverted gravity column

```
gravity reset on
G = cartGrid([1, 1, 40], [1, 1, 10]);
G = computeGeometry(G);
rock = makeRock(G, 0.1*darcy, 1);
fluid = initCoreyFluid(
    'mu', [0.30860, 0.056641]*centi*poise,
    'rho', [975.86, 686.54]*kilogram/meter^3,
    'n', [2,2], 'sr', [.1,.2], 'kwm',[.2142,.85]);
hT = computeTrans(G, rock);

xr = initResSol(G, 100.0*barsa, 1.0);
xr.s(end/2+1:end) = 0.0;

xr = incompTPFA(xr, G, hT, fluid);
dt = 5*day; t=0;
for i=1:150
    xr = explicitTransport(xr, G, dt, rock, fluid, 'onlygrav', true);
    t = t+dt;
    xr = incompTPFA(xr, G, hT, fluid);
end
```
Example: inverted gravity column

$t = 0$ days

$t = 250$ days

$t = 500$ days

$t = 125$ days

$t = 375$ days

$t = 750$ days
Potential pitfall: capillary-dominated flow
1 Introduction
2 Getting started with MRST
3 Grids and petrophysical data
4 Incompressible flow
5 Multiphase flow
6 Compressible flow
7 The AD-OO framework in MRST
What you will learn in this section

In this section, we study compressible single-phase and multiphase flow. You will learn about:

- rapid prototyping of new models using AD
- use of discrete operators for compact implementations
- the AD-OO framework

To learn more:
- study examples/tutorials in the ad-core and ad-blackoil modules
- read Chapter 9 in the MRST book
- read Krogstad et al. (SPE RSS, 2015), doi: 10.2118/173317-MS
- read Bao et al. (COMG, 2017), doi: 10.1007/s10596-017-9624-5
The governing equation is

\[ c \frac{\partial p}{\partial t} - \nabla \cdot \left( \frac{K}{\mu} \nabla p \right) = 0 \]

Semi-discrete flow equations on residual form with implicit time discretization and discrete operators $\text{div}$, $\text{grad}$.

\[ \frac{1}{\Delta t} c(p^{n+1} - p^n) - \text{div} \left( \frac{K}{\mu} \text{grad}(p) \right)^{n+1} = 0 \]

```matlab
presEq = @(p, p0, dt) (1/dt)*c*(p-p0) - div( (T/mu).*grad(p));
```

**current time step**  **previous time step**
Single-phase weakly compressible flow

```
load seamount
G = pebi(triangleGrid([x(:) y(:)]));
G.nodes.coords = G.nodes.coords*100;
:
c = 1e-4;
mu = 1*centi*poise;

presEq = @(p, p0, dt) ...
    (1/dt)*c*(p-p0) - div( (T/mu).*grad(p));

p0 = 100*atm*ones(nc, 1); p0(r<5) = 200*atm;
p = initVariablesADI(p0);
[t,T,dt] = deal(0,10*day,hour);
while t < T,
    t = t + dt;
    p0 = p.val;

    eq = presEq(p, p0, dt);
    p.val = p.val - (eq.jac{1} \ eq.val);

    clf, plotCellData(G,p.val);
    caxis([100 200]*atm); drawnow;
end
```
Single-phase compressible flow

Weakly compressible model:

\[ c \frac{\partial p}{\partial t} - \nabla \cdot (K \nabla p) = 0 \]

% Fluid properties
\texttt{c = 1e-4;}
\texttt{mu = 1*centi*poise;}

% Set up equation
\texttt{presEq = @(p, p0, dt) ...}
\texttt{(1/dt)*c*(p-p0) - div((T/mu).*grad(p));}

Model with rock and fluid compressibility:

\[ \frac{\partial}{\partial t} (\phi \rho) - \nabla \cdot (\rho K \nabla p) = 0 \]

% Rock property
\texttt{[phi0,c_r,pr] = deal(0.3, 1e-3, 1*atm);}
\texttt{phi = @(p) ... phi0 + (1-phi0)*(1-exp(-c_r*(p-pr)));

% Fluid properties
\texttt{rho0 = 10^3; c_f = 5e-5;}
\texttt{rho = @(p) (rho0*exp(c_f*(p - pref)));
\texttt{mu = 1*centi*poise;}

% Set up equation
\texttt{pv = @(p) (phi(p).*G.cells.volumes );}
\texttt{presEq = @(p, p0, dt) ...}
\texttt{(1/dt)*(pv(p).*rho(p) - pv(p0).*rho(p0)) ...}
\texttt{-div(avg(rho(p)).*(T/mu).*grad(p));}

avg is a face-average operator: \( \mathbb{R}^{n_c} \rightarrow \mathbb{R}^{n_f} \)
Adding effects: gravity

Semi-discrete flow equations on residual form:

\[
\frac{1}{\Delta t}[(\phi\rho)^{n+1}-(\phi\rho)^n]+\text{div}((\rho v)^{n+1})=q, \quad v = -\frac{K}{\mu}(\text{grad}(p)-g\rho \text{grad}(z))
\]

Homogeneous equation implemented in MRST

```matlab
gradz = grad(G.cells.centroids(:,3));
v = @(p) -(T/mu).* ( grad(p) - g*avg(rho(p)).*gradz );
presEq = @(p, p0, dt) (1/dt)*(pv(p).*rho(p) - pv(p0).*rho(p0)) ... + div( avg(rho(p)).*v(p));
```
Adding effects: gravity

Semi-discrete flow equations on residual form:

\[
\frac{1}{\Delta t} [(\phi \rho)^{n+1} - (\phi \rho)^n] + \text{div} (\rho v)^{n+1} = q, \quad v = -\frac{K}{\mu} (\text{grad}(p) - g \rho \text{grad}(z))
\]

Homogeneous equation implemented in MRST

\begin{verbatim}
gradz = grad(G.cells.centroids(:,3));
v = @(p) -T/mu.*( grad(p) - g*avg(rho(p)).*gradz );

presEq = @(p, p0, dt) (1/dt)*(...
    + div( avg(rho(p)).*v(p));
\end{verbatim}

\begin{itemize}
  \item current time step
  \item previous time step
  \item \( \rho \) at cell face
\end{itemize}
Peacemann well model, with hydrostatic pressure in well bore, and control on bottom-hole pressure:

\[ p_c = p_{bh} + g \Delta z_c \rho (p_{bh}), \]

\[ q_c = \frac{\rho}{\mu} WI (p_c - p), \]

\[ q^S = \frac{1}{\rho^S} \sum_c q_c, \]

\[ p_{bh} = \text{constant} \]

Implemented in MRST:

\[ wc = W(1).\text{cells}; \quad \% \text{connection grid cells} \]
\[ WI = W(1).WI; \quad \% \text{well--indices} \]
\[ dz = W(1).dZ; \quad \% \text{connection depth relative to bottom--hole} \]

\[ p_{\text{conn}} = @bhp \quad \text{bhp + g*dz*rho(bhp)}; \]
\[ q_{\text{conn}} = @p(bhp) \quad \text{WI*(rho(p(wc))/mu)*(p_{\text{conn}}(bhp) - p(wc))}; \]
\[ \text{rateEq} = @p(bhp, qS) \quad qS - \text{sum(q_{\text{conn}}(p, bhp))/rhoS}; \]
\[ \text{ctrlEq} = @(bhp) \quad \text{bhp - 100*barsa}; \]
Details of simulator: time loop and assembly of equations

```matlab
[p, bhp, qS] = ...
    initVariablesADI(pin, pin(wc(1)), 0);
t = 0; step = 0;
while t < totTime,
    t = t + dt;

    % Newton loop
    resNorm = 1e99;
p0 = double(p); % Previous step pressure
nit = 0;
while (resNorm > tol) && (nit < maxits)
    % one Newton iteration
    end

    if nit > maxits,
        error('Newton solves did not converge')
    end
end
```
Details of simulator: time loop and assembly of equations

```matlab
[p, bhp, qS] = ...  
initVariablesADI(p);

t = 0; step = 0;
while t < totTime,
    t = t + dt;
    % Newton loop
    resNorm = 1e99;
    p0 = double(p); % Previous step pressure
    nit = 0;
    while (resNorm > tol) && (nit < maxits)
        % one Newton iteration
        end
    end
    if nit > maxits,
        error('Newton solves did not converge')
    end
end

% -- ONE NEWTON ITERATION
% Add source terms to homogeneous pressure equation:
eq1  = presEq(p, p0, dt);
eq1(wc) = eq1(wc) - q_conn(p, bhp);

% Collect all equations
eqs = {eq1, rateEq(p, bhp, qS), ctrlEq(bhp)};

% Concatenate equations and solve for update:
eq  = cat(eqs{:});
J   = eq.jac{1}; % Jacobian
res = eq.val;   % residual
upd = -(J \ res); % Newton update

% Update variables
p.val  = p.val + upd(pIx);
bhp.val = bhp.val + upd(bhpIx);
qS.val = qS.val + upd(qSIX);

resNorm = norm(res);
nit    = nit + 1;
```
Adding effects: pressure-dependent viscosity

Assume the following model:

\[ \mu(p) = \mu_0 \left[ 1 + c_r (p - p_r) \right] \]
Adding effects: pressure-dependent viscosity

Assume the following model:

\[ \mu(p) = \mu_0 [1 + c_r (p - p_r)] \]

Arithmetic averaging:

\[
\begin{align*}
\text{mu} &= @(p) \mu_0 * (1 + c_\mu * (p - p_r)); \\
\text{v} &= @(p) - (T./\mu(\text{avg}(p))).*(\text{grad}(p) - g*\text{avg}(\rho(p)).*\text{dz}); \\
\text{qcon} &= @(p, \text{bhp}) \ \text{WI}.*(\rho(p(wc))./\mu(p(wc))).*(\text{pcon}(\text{bhp}) - p(wc));
\end{align*}
\]

This is all! No need to recompute derivatives for Newton’s method

Unfortunately, this approach is only correct on Cartesian grids
Adding effects: pressure-dependent viscosity

Assume the following model:

\[ \mu(p) = \mu_0 [1 + c_r (p - p_r)] \]

Harmonic averaging:

\[
[cn,F] = \text{getCellNoFaces}(G); \\
hf2f = \text{sparse}(F,(1: \text{numel}(cn))^1,1); \\
hf2f = hf2f( \text{all}(C\sim=0,2),:); \\
fmob = @(mu,p) 1./(hf2f*(mu(p(cn))./hT))
\]

\[ v = @p( -fmob(mu,p).* (grad(p) - g*avg(rh0(p)).*dz)); \]

We multiply each one-sided transmissibility \( T_{i,k} \) by the correct \( \mu(p) \) value and then compute their harmonic average.

Previously, we used \text{accumarray} to average \( T_{i,k} \), but this function does not work for AD variables. Hence, we multiply by a sparse matrix instead.
Adding effects: thermal flow

\[
\frac{\partial}{\partial t} [\phi \rho(p, T)] + \nabla \cdot [\rho(p, T) \vec{v}] = q, \quad \vec{v} = -\frac{K}{\mu(p, T)} \left[ \nabla p - g \rho(p, T) \nabla z \right]
\]

\[
\frac{\partial}{\partial t} \left[ \phi \rho(p, T) E_f(p, t) + (1 - \phi) E_r(p, T) \right] + \nabla \cdot \left[ \rho(p, T) H_f(p, T) \vec{v} \right] - \nabla \cdot [\kappa \nabla T] = q_e
\]

Constitutive laws and operators

\[
pv = \text{poreVolume}(G, \text{rock});
\]

\[
\rho = \text{@}(p, T) \times rhor \times (1 + (cp \times (p - pr))) \times \exp(-ct \times (T - Tr));
\]

\[
\mu = \text{@}(p, T) \mu0 \times (1 + cmup \times (p - p_r)) \times \exp(-cmut \times (T - T_r));
\]

\[
H_f = \text{@}(p, T) \times Cw \times T + (1 - Tr \times ct) \times (p-pr) \times \rho(p, T); \\
\]

\[
E_f = \text{@}(p, T) \times H_f(p, T) - p \times \rho(p, T); \\
\]

\[
Er = \text{@}(T) \times Cr \times T; \\
\]

\[
upw = \text{@}(x, flag) \times x(N(:,1)) \times \text{double}(flag) ... \\
+ x(N(:,2)) \times \text{double}(`flag); \\
\]

Discrete equations

\[
v = \text{@}(p, T) - (Tr \times \text{mu}(\text{avg}(p), \text{avg}(T))) ... \\
\times (\text{grad}(p) - g \times \text{avg}(\rho(p, T)) \times dz );
\]

\[
pEq = \text{@}(p, T, p0, T0, dt) ... \\
(1/dt) \times (pv(p) \times \rho(p0, T) - pv(p0) \times \rho(p0, T0)) ... \\
+ \text{div}( \text{avg}(\rho(p, T)) \times v(p, T) );
\]

\[
hEq = \text{@}(p, T, p0, T0, dt) ... \\
(1/dt) \times (pv(p) \times \rho(p, T) \times Ef(p, T) + spv(p) \times Er(T) ... \\
- pv(p0) \times \rho(p0, T0) \times Ef(p0, T0) - spv(p0) \times Er(T0)) ... \\
+ \text{div}( \text{upw}(Hf(p, T), v(p, T) > 0) \times \text{avg}(\rho(p, T)) \times v(p, T) ) ... \\
+ \text{div}( -Th \times \text{grad}(T));
\]
Adding effects: thermal flow

\[
\frac{\partial}{\partial t} \left[ \phi \rho(p, T) \right] + \nabla \cdot \left[ \rho(p, T) \vec{v} \right] = q, \quad \vec{v} = -\frac{K}{\mu(p, T)} \left[ \nabla p - g \rho(p, T) \nabla z \right]
\]

\[
\frac{\partial}{\partial t} \left[ \phi \rho(p, T) E_f(p, t) + (1 - \phi) E_r(p, T) \right] + \nabla \cdot \left[ \rho(p, T) H_f(p, T) \vec{v} \right] - \nabla \cdot \left[ \kappa \nabla T \right] = q_e
\]

Constitutive laws and operators

```matlab
pvr = poreVolume(G, rock);
pv = @(p) pvr .* exp( cr * (p - pr) );
spv = @(p) G.cells.volumes - pv(p);

rho = @(p,T) rhor.*(1+(cp*(p - pr))).*exp(-ct*(T-Tr));
mu = @(p,T) mu0*(1+cmup*(p-p_r)).*exp(-cmut*(T-T_r));

Hf = @(p,T) Cw*T + (1-Tr*ct).*(p-pr)./rho(p,T);
Ef = @(p,T) Hf(p,T) - p./rho(p,T);
Er = @(T) Cr*T;

upw = @(x,flag) x(N(:,1)).*double(flag) ... + x(N(:,2)).*double(~flag);
```

Discrete equations

```matlab
v = @(p,T) -(Tr./mu(avg(p),avg(T))) .* ( grad(p) - g*avg(rho(p,T)).*dz );
pEq = @(p,T, p0, T0, dt) ... (1/dt)*(pv(p).*rho(p,T) - pv(p0).*rho(p0,T0)) ... + div( avg(rho(p,T)).*v(p,T) );

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

Anonymous functions may lead to redundant function evaluations. To cure, move computation of residuals inside a function and compute and store constitutive relationships in temporary variables.
Adding effects: multiple phases (without mass transfer)

\[
\frac{(\phi S\alpha \rho\alpha)^{n+1} - (\phi S\alpha \rho\alpha)^n}{\Delta t^n} + \text{div}(\rho v)^{n+1}_\alpha = (\rho q)^{n+1}_\alpha \\
\nu^{n+1}_\alpha = -\frac{K k_{r\alpha}}{\mu^{n+1}_\alpha} \left[ \text{grad}(p^{n+1}) - g \rho^{n+1}_\alpha \text{grad}(z) \right]
\]
Adding effects: multiple phases (without mass transfer)

\[
\frac{(\phi S_\alpha \rho_\alpha)^{n+1} - (\phi S_\alpha \rho_\alpha)^n}{\Delta t^n} + \text{div}(\rho v)^{n+1}_\alpha = (\rho q)^{n+1}_\alpha
\]

\[
v^{n+1}_\alpha = -\frac{K k_{r\alpha}}{\mu^{n+1}_\alpha} \left[ \text{grad}(p^{n+1}) - g\rho^{n+1}_\alpha \text{grad}(z) \right]
\]

We start by computing all cell-based properties:

% Densities and pore volumes
[rW, rW0, rO, rO0] = deal(rhoW(p), rhoW(p0), rhoO(p), rhoO(p0));
[vol, vol0] = deal(pv(p), pv(p0));

% Mobility: Relative permeability over constant viscosity
mobW = krW(sW)./muW;
mobO = krO(1-sW)./mu0;
Adding effects: multiple phases (without mass transfer)

\[
\frac{(\phi S_\alpha \rho_\alpha)^{n+1} - (\phi S_\alpha \rho_\alpha)^n}{\Delta t^n} + \operatorname{div}(\rho v_\alpha)^{n+1} = (\rho q_\alpha)^{n+1}
\]

\[
v_\alpha^{n+1} = - \frac{K k_{r\alpha}}{\mu_\alpha^{n+1}} \left[ \operatorname{grad}(p^{n+1}) - g \rho_\alpha^{n+1} \operatorname{grad}(z) \right]
\]

Next, we compute differences in phase pressure across cell interfaces:

```matlab
dp = grad(p);
dpW = dp - g * avg(rW) .* gradz;
dp0 = dp - g * avg(r0) .* gradz;
```

and use this to define upwind-weighted fluxes:

```matlab
upw = @(flag, x) flag.*x(C(:, 1)) + ~flag.*x(C(:, 2));

vW = -upw(double(dpW) <= 0, rW.*mobW).*T.*dpW;
v0 = -upw(double(dp0) <= 0, r0.*mob0).*T.*dp0;
```
Adding effects: multiple phases (without mass transfer)

\[
\frac{(\phi S_\alpha \rho_\alpha)^{n+1} - (\phi S_\alpha \rho_\alpha)^n}{\Delta t^n} + \text{div}(\rho v)^{n+1}_\alpha = (\rho q)^{n+1}_\alpha
\]

\[
v^{n+1}_\alpha = - \frac{K k_{r\alpha}}{\mu^{n+1}_\alpha} \left[ \text{grad}(p^{n+1}) - g \rho^{n+1}_\alpha \text{grad}(z) \right]
\]

Now, we have all we need to compute residual equations

\[
\text{water} = \frac{1}{dt(n)} \times (\text{vol.} \times rW \times sW - \text{vol0.} \times rW0 \times sW0) + \text{div}(vW);
\]

\[
\text{oil} = \frac{1}{dt(n)} \times (\text{vol.} \times rO \times (1-sW) - \text{vol0.} \times rO0 \times (1-sW0)) + \text{div}(vO);
\]

\[
eqs = \{\text{oil, water}\};
\]

\[
eq = \text{cat(eqs{}});\]
Adding effects: multiple phases (without mass transfer)

\[
\frac{(\phi S_{\alpha} \rho_{\alpha})^{n+1} - (\phi S_{\alpha} \rho_{\alpha})^{n}}{\Delta t^n} + \text{div}(\rho v)^{n+1}_\alpha = (\rho q)^{n+1}_\alpha
\]

\[
v^{n+1}_\alpha = -\frac{K k_{r \alpha}}{\mu^{n+1}_\alpha} \left[ \text{grad}(p^{n+1}) - g \rho^{n+1}_\alpha \text{grad}(z) \right]
\]

To get a robust simulator, we would also need to include:

- Time-step control inside the loop
- A line-search algorithm rather than simple Newton
- Possibly also some preconditioning method

Notice also that this code cannot be used to simulate *incompressible flow*. Trick: add small rock compressibility.
Rapid prototyping in MRST

- Use abstractions to express your ideas in a form close to the underlying mathematics

- The interactive environment offers you:
  - ability to try out each operation and build program as you go
  - wide range of built-in functions for numerical computations
  - powerful data analysis, graphical user interface, and visualization

- Easy to debug and modify/improve existing codes:
  - run code line by line, inspect and change variables at any point
  - step back and rerun parts of code with changed parameters
  - add new behavior and data members while executing program

- Later, one can, if necessary, replace bottleneck operations with accelerated editions implemented in a compiled language
1 Introduction

2 Getting started with MRST

3 Grids and petrophysical data

4 Incompressible flow

5 Multiphase flow

6 Compressible flow

7 The AD-OO framework in MRST
So far in the lecture, we have seen how automatic differentiation can be used to prototype simulators. Writing a single script has advantages:

- fast to prototype
- self-contained and easy to modify

However, there are some disadvantages as well:

- mixing logic of Newton solver with definition of model equations
- time-stepping and plotting will be done per script
- implementing several variations of the same model will result in code duplication
Advanced simulators: motivation

Code will eventually start to become complicated:

- complex rock-fluid/PVT models
- hysteretic behavior (post-iteration updates)
- wells with advanced schedules and controls
- time-step control and iteration control
- CPR type preconditioners and multigrid solvers
- advanced flow models that are extensions of simpler models
- sub-equations with different discretizations, nested iterations, ...
Next step: object-orientation

Introduce object-orientation to separate:
- physical models
- discretizations and discrete operators
- nonlinear solver and time-stepping
- assembly and solution of the linear system

Only expose needed details and enable more reuse of functionality that has already been developed
Next step: object-orientation

The object-oriented AD framework makes it easy to write general simulator classes:

- standardized interfaces make Newton solver independent of the specifics of the physical model
- standardized input/output makes it easy to compare and plot results
- switching linear solvers or time-stepping strategy is straightforward
  (Compare ad-blackoil and blackoil-sequential)

Typical workflow: build simple prototype → migrate to class-based solver
The AD-OO modules

<table>
<thead>
<tr>
<th>MRST core</th>
<th>Basic functions/data structures: grid, petrophysics, wells, boundary conditions, I/O, grid processing, AD library, plotting, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>ad-core</td>
<td>General simulation framework: abstract model classes, time-step and iteration control, linearizations, linear solvers, hooks for I/O and plotting, ...</td>
</tr>
<tr>
<td>mrst-gui</td>
<td>Graphical interfaces for interactive visualization of reservoir states and petrophysical data</td>
</tr>
<tr>
<td>deckformat</td>
<td>Input of ECLIPSE simulation decks: read, convert to SI units, and construct MRST objects for grids, fluid and rock properties, wells and simulation schedules</td>
</tr>
<tr>
<td>ad-blackoil</td>
<td>General 3-phase black-oil simulator with dissolution and vaporization, specialized 1- and 2-phase models, CPR preconditioning</td>
</tr>
<tr>
<td>ad-props</td>
<td>Initialization of fluid models from ECLIPSE input decks</td>
</tr>
<tr>
<td>ad-eor</td>
<td>Fully implicit simulators for water-based EOR: polymer and surfactant</td>
</tr>
</tbody>
</table>
Example: two-phase Buckley–Leverett

```matlab
G = cartGrid([50, 1, 1], [1000, 10, 10]*meter);
G = computeGeometry(G);
rock = makeRock(1*darcy*ones, .3);
fluid = initSimpleADIFluid('phases', 'WO', 'n', [2 2]);

% Set up model and initial state.
model = TwoPhaseOilWaterModel(G, rock, fluid);
state0 = initResSol(G, 50*barsa, [0, 1]);
state0.wellSol = initWellSolAD([], model, state0);

% Set up drive mechanism: constant rate at x=0, constant pressure at x=L
injr = -sum(poreVolume(G,rock))/(500*day);
bc = fluxside([], G, 'xmin', -injr, 'sat', [1, 0]);
bc = pside(bc, G, 'xmax', 0*barsa, 'sat', [0, 1]);
```
Example: two-phase Buckley–Leverett

```matlab
G = cartGrid([50, 1, 1], [1000, 10, 10]*meter);
G = computeGeometry(G);
rock = makeRock(1*darcy*ones, .3);
fluid = initSimpleADIFluid('phases', 'WO', 'n', [2 2]);

% Set up model and initial state.
model = TwoPhaseOilWaterModel(G, rock, fluid);
state0 = initResSol(G, 50*barsa, [0, 1]);
state0.wellSol = initWellSolAD([], model, state0);

% Set up drive mechanism: constant rate at x=0, constant pressure at x=L
injR = -sum(poreVolume(G, rock))/(500*day);
bc = fluxside([], G, 'xmin', -injR, 'sat', [1, 0]);
bc = pside(bc, G, 'xmax', 0*barsa, 'sat', [0, 1]);
```

Simulate 1 PVI using a manual loop:

```matlab
[dT, n] = deal(20*day, 25);
states = cell(n+1, 1);
states{1} = state0;
solver = NonLinearSolver();

for i = 1:n
    state = solver.solveTimestep(states{i}, dT, model, 'bc', bc);
    states{i+1} = state;
end
```

..
Example: two-phase Buckley–Leverett

```matlab
G = cartGrid([50, 1, 1], [1000, 10, 10]*meter);
G = computeGeometry(G);
rock = makeRock(1*darcy*ones, .3);
fluid = initSimpleADI Fluid('phases', 'WO', 'n', [2 2]);

% Set up model and initial state.
model = TwoPhaseOilWaterModel(G, rock, fluid);
state0 = initResSol(G, 50*barsa, [0, 1]);
state0.wellSol = initWellSolAD([], model, state0);

% Set up drive mechanism: constant rate at x=0, constant pressure at x=L
injR = -sum(poreVolume(G, rock))/(500*day);
bc = fluxside([], G, 'xmin', -injR, 'sat', [1, 0]);
bc = pside(bc, G, 'xmax', 0*barsa, 'sat', [0, 1]);

Simulate 1 PVI using a manual loop:

```matlab
[dT, n] = deal(20*day, 25);
states = cell(n+1, 1);
states{1} = state0;
solver = NonLinearSolver();

for i = 1:n
    state = solver.solveTimestep(states{i}, dT, model, 'bc', bc);
    states{i+1} = state;
end
plotToolbar(G, states, 'field', 's:1', 'plot1d', true, ...
            'lockCaxis', true, 'startplayback', true);
```
Example: two-phase Buckley–Leverett

```matlab
G = cartGrid([50, 1, 1], [1000, 10, 10]*meter);
G = computeGeometry(G);
rock = makeRock(1+darcy*ones, .3);
fluid = initSimpleADIFluid('phases', 'WO', 'n', [2 2]);

% Set up model and initial state.
model = TwoPhaseOilWaterModel(G, rock, fluid);
state0 = initResSol(G, 50*barsa, [0, 1]);
state0.wellSol = initWellSolAD([], model, state0);

% Set up drive mechanism: constant rate at x=0, constant pressure at x=L
injR = -sum(poreVolume(G, rock))/(500*day);
bc = fluxside([], G, 'xmin', -injR, 'sat', [1, 0]);
bc = pside(bc, G, 'xmax', 0*barsa, 'sat', [0, 1]);
```

Repeat simulation with general solver:

```matlab
schedule = simpleSchedule(repmat(dT,1,25), 'bc', bc);
[~, sstates] = simulateScheduleAD(state0, model, schedule);

plotToolbar(G, sstates, 'field', 's:1', 'lockCaxis', true),
caxis([0 1]), view(10,10)
colorbar
```
Example: two-phase Buckley–Leverett

```matlab
G = cartGrid([50, 1, 1], [1000, 10, 10]*meter);
G = computeGeometry(G);
rock = makeRock(1*darcy*ones, .3);
fluid = initSimpleADIFluid('phases', 'WO', 'n', [2 2]);

% Set up model and initial state.
model = TwoPhaseOilWaterModel(G, rock, fluid);
state0 = initResSol(G, 50*barsa, [0, 1]);
state0.wellSol = initWellSolAD([], model, state0);

% Set up drive mechanism: constant rate at x=0, constant pressure at x=L
injR = -sum(poreVolume(G, rock))/(500*day);
bc = fluxside([], G, 'xmin', -injR, 'sat', [1, 0]);
bc = pside(bc, G, 'xmax', 0*barsa, 'sat', [0, 1]);
```

The general solver has a hook, that visualizes the progress of the simulation, enables you to stop it and continue running in 'debug' mode:

```matlab
fn = getPlotAfterStep(state0, model, schedule, ...
    'plotWell', false, 'plotReservoir', true, 'field', 's:1', ...
    'lockCaxis',true, 'plot1d', true);

[~,sstates,report] = ...
    simulateScheduleAD(state0, model, schedule,'afterStepFn', fn);
```
More about the nonlinear solver

**Main loop**

- $[x, t] = \text{initializeSolution}(...)\$
- while $t < T$
  - $t = t + dt$
  - $x = xit$
  - do
    - $[R, J] = \text{computeResiduals}(xit, x)$
    - $\text{upd} = J^{-1}R$
    - $xit = xit + \text{upd}$
    - while $\|R\| > \text{tol}$
      - end
  - end

**solveMinistep**

- $[\text{res}, J, \ldots] = \text{getEqs}(t + \tau, \ldots)$
- $xit = x$
- while $\text{res} > \text{tol} \& \text{it} \leq \text{itmax}$
  - $lsys = \text{assembleLinSys}(\text{res, J, \ldots})$
  - $\text{upd} = \text{solveLinSys}(\text{xit, lsys, lsol, \ldots})$
  - $\text{upd} = \text{stabilizeStep}(\text{xit, upd, lsys, \ldots})$
  - $\text{xit} = \text{updateIterate}(\text{upd, \ldots})$
  - $\text{cleanupLinSolver}(\text{lsol})$
  - $[\text{res, J}] = \text{getEqs}(t + \tau, \ldots)$
  - end
  - if $\text{it} \leq \text{itmax}$
    - $\text{ok} = \text{true}$
    - $[\tau, x, \ldots] = \text{updateSolution}(\text{xit})$
  - else
    - $\text{ok} = \text{false}$
  - end

**Context:**
- physical model and reservoir state
- nonlinear solver and time loop
- linearization of discrete equations
- linear solver

**Classic Newton**

- $[\text{xit, t}] = \text{initializeSolution}(...)$
- while $t < T$
  - $t = t + dt$
  - $x = xit$
  - do
    - $[R, J] = \text{computeResiduals}(\text{xit, x})$
    - $\text{upd} = J^{-1}R$
    - $\text{xit} = xit + \text{upd}$
    - while $\|R\| > \text{tol}$
      - end
  - end
More about the nonlinear solver

Nonlinear solver
Solves nonlinear problems sub-divided into one or more *mini steps* using Newton’s method

Initial ministep: \( \Delta t \)

Time step selector
Determines optimal time steps

\[ \text{SimpleTimeStepSelector, IterationCountSelector, StateChangeTimeStepSelector, ...} \]

Adjusted: \( \Delta \tilde{t} \)

Primary vars
\[ [\text{Res, Jac}], \text{info} \]

Assemble:
\[ A\delta x = b \]

Update variables:
\[ p \leftarrow p + \delta p, s \leftarrow s + \delta s, ... \]

Well solutions

Well data:
\[ q_W, q_O, q_G, bhp, ... \]

Physical model
Defines mathematical model: Residual equations, Jacobians, limits on updates, convergence definition...

\[ \text{TwoPhaseOilWaterModel, ThreePhaseBlackOilModel} \]

Well model
Well equations, control switch, wellbore pressure drop, ...

Linearized problem
Jacobians, residual equations and meta-information about their types

Linear solver
Solves linearized problem and returns increments

\[ \text{BackslashSolverAD, AGMGSolverAD, CPRSolverAD, MultiscaleSolverAD, ...} \]

Assemble:
\[ A x = b \]

Type color legend

Class

Struct

Input

Contains object
The framework is designed so that you can only work on the components you are interested in: If you want to write a flow solver, you do not need to debug a Newton solver.
Functionality through inheritance

**PhysicalModel**
- Abstract base class for all MRST models.
- Contains logic related to linearization and updates.
- Primary variables: None

**ReservoirModel**
- Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature.
- Base class for all reservoir models.
- Added primary variables: \( s_\alpha, p, T, q_\alpha, p_{bh} \)

**ThreePhaseBlackOilModel**
- Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.
- Added primary variables: \( r_s, r_v \)
Functionality through inheritance

**PhysicalModel**

Abstract base class for all MRST models. Contains logic related to linearization and updates.

Primary variables: None

**ReservoirModel**

Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature. Base class for all reservoir models.

Added primary variables: $s_\alpha, p, T, q_\alpha, p_{bh}$

**ThreePhaseBlackOilModel**

Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.

Added primary variables: $r_s, r_v$

**Properties:**

- operators, G
- nonlinearTolerance, stepFunctionIsLinear
- verbose
Functionality through inheritance

PhysicalModel
Abstract base class for all MRST models.
Contains logic related to linearization and updates.

Primary variables: None

ReservoirModel
Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature.
Base class for all reservoir models.

Added primary variables: \( s_\alpha, p, T, q_\alpha, p_{bh} \)

ThreePhaseBlackOilModel
Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.

Added primary variables: \( r_s, r_v \)

PhysicalModel
Properties:
- operators, G
- nonlinearTolerance, stepFunctionIsLinear
- verbose

Quality assurance:
- \( state = model.validateState(state) \)
- \( model = model.validateModel(...) \)
### Functionality through inheritance

**Abstract base class for all MRST models.**
Contains logic related to linearization and updates.

**Primary variables:** None

---

**ReservoirModel**
Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature.
Base class for all reservoir models.

**Added primary variables:** \( s_\alpha, p, T, q_\alpha, p_{bh} \)

---

**ThreePhaseBlackOilModel**
Extends ReservoirModel with optional solution gas and vaporized oil.
Base class for two- and single-phase versions.

**Added primary variables:** \( r_s, r_v \)

---

**PhysicalModel**

**Properties:**
- operators, \( G \)
- nonlinearTolerance, stepFunctionIsLinear
- verbose

**Quality assurance:**
- \( \text{state} = \text{model}.\text{validateState}(\text{state}) \)
- \( \text{model} = \text{model}.\text{validateModel}(\ldots) \)

**Querying / setting model properties:**
- \( p = \text{model}.\text{getProp}(\text{state}, '\text{pressure}') \)
- \( [p,s] = \text{model}.\text{getProps}(\text{state}, '\text{pressure}', 's') \)
- \( [f,i] = \text{model}.\text{getVariableField}(\text{name}) \)
- \( \text{state} = \text{model}.\text{setProp}(\text{model}, \text{state}, '\text{pressure}', 5) \)
- \( \text{state} = \text{model}.\text{incrementProp}(\text{state}, '\text{pressure}', 1) \)
- \( \text{state} = \text{model}.\text{capProperty}(\text{state}, '\text{saturation}', 0, 1) \)

These are examples of syntax for derived classes and will not work on a PhysicalModel, which has no associated variables.
Functionality through inheritance

---

**PhysicalModel**
Abstract base class for all MRST models. Contains logic related to linearization and updates.

Primary variables: None

---

**ReservoirModel**
Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature. Base class for all reservoir models.

Added primary variables: $s_\alpha, p, T, q_\alpha, p_{bh}$

---

**ThreePhaseBlackOilModel**
Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.

Added primary variables: $r_s, r_v$

---

**Get drive mechanisms:**

$[\ldots, ctrl] = \text{model.getDrivingForces}(\text{model, ctrl})$
Functionality through inheritance

<table>
<thead>
<tr>
<th>PhysicalModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract base class for all MRST models. Contains logic related to linearization and updates.</td>
</tr>
<tr>
<td>Primary variables: None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReservoirModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature. Base class for all reservoir models.</td>
</tr>
<tr>
<td>Added primary variables: $s_\alpha, p, T, q_\alpha, p_{bh}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ThreePhaseBlackOilModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.</td>
</tr>
<tr>
<td>Added primary variables: $r_s, r_v$</td>
</tr>
</tbody>
</table>

**PhysicalModel**

Get drive mechanisms:

```plaintext
[...ctrl] = model.getDrivingForces(model, ctrl)
```

Linearize and assemble discrete problem:

```plaintext
[problem, state] = ...
model.getEquations(state0, state, ...
dt, drivingForces, varargin)
```
## Functionality through inheritance

### PhysicalModel

- **Abstract base class for all MRST models.**
- **Contains logic related to linearization and updates.**
- **Primary variables:** None

### ReservoirModel

- **Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature.**
- **Base class for all reservoir models.**
- **Added primary variables:** $s_\alpha$, $p$, $T$, $q_\alpha$, $p_{bh}$

### ThreePhaseBlackOilModel

- **Extends ReservoirModel with optional solution gas and vaporized oil.**
- **Base class for two- and single-phase versions.**
- **Added primary variables:** $r_s$, $r_v$

### Get drive mechanisms:

```
[... , ctrl] = model.getDrivingForces(model, ctrl)
```

### Linearize and assemble discrete problem:

```
[problem, state] = ... 
model.getEquations(state0, state, ... 
    dt, drivingForces, varargin)
```

### Compute a linearized time step:

```
[state, report] = ... 
model.stepFunction(model, state, state0, ... 
    dt, drivingForces, linsolve, ... 
    nonlinsolve, iteration, varargin)
```
Functionality through inheritance

<table>
<thead>
<tr>
<th>PhysicalModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract base class for all MRST models. Contains logic related to linearization and updates.</td>
</tr>
<tr>
<td>Primary variables: None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReservoirModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature. Base class for all reservoir models.</td>
</tr>
<tr>
<td>Added primary variables: $s_\alpha, p, T, q_\alpha, p_{bh}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ThreePhaseBlackOilModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.</td>
</tr>
<tr>
<td>Added primary variables: $r_s, r_v$</td>
</tr>
</tbody>
</table>

**PhysicalModel**

Update state from Newton increment:

```
[state, report] = model.updateState(state, ...  
  problem, dx, drivingForces)
```

and other utility functions:

```
[conv, ..] = model.checkConvergence(problem, n)  
[state,rep] = model.updateAfterConvergence(...  
  state0, state, dt, drivingForces)  
  :  
  :
```
Functionality through inheritance

### PhysicalModel

Abstract base class for all MRST models. Contains logic related to linearization and updates.

**Primary variables:** None

### ReservoirModel

Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature. Base class for all reservoir models.

**Added primary variables:** \( s_\alpha, p, T, q_\alpha, p_{bh} \)

### ThreePhaseBlackOilModel

Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.

**Added primary variables:** \( r_s, r_v \)

### Properties:

**% Submodels**
- fluid, rock, gravity

**FacilityModel**

**% Physical properties**
- water, gas, oil
- saturationVarNames, componentVarNames

**% Iterations parameters**
- dpMaxRel, dpMaxAbs, dsMaxRel, dsMaxAbs
- maximumPressure, minimumPressure
- useCNVConvergence, toleranceCNV, toleranceMB

**% Miscellaneous**


### Functionality through inheritance

#### PhysicalModel

Abstract base class for all MRST models. Contains logic related to linearization and updates.

Primary variables: None

#### ReservoirModel

Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature. Base class for all reservoir models.

Added primary variables: \( s_\alpha, p, T, q_\alpha, p_{bh} \)

#### ThreePhaseBlackOilModel

Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.

Added primary variables: \( r_s, r_v \)

---

```matlab
function [fn,ix] = getVariableField(model, name)
    switch(lower(name))
      case {'pressure', 'p'}
        ix = 1;
        fn = 'pressure';
      case {'s', 'sat', 'saturation'}
        ix = '.1';
        fn = 's';
      case {'sw', 'water'}
        ix = model.satVarIndex('sw');
        fn = 's';
      :
    end
```

Plus a large number of utility functions to extract, update, and store these physical variables.
### Functionality through inheritance

**PhysicalModel**

Abstract base class for all MRST models. Contains logic related to linearization and updates.

Primary variables: None

**ReservoirModel**

Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature. Base class for all reservoir models.

Added primary variables: \(s_\alpha, p, T, q_\alpha, p_{bh}\)

**ThreePhaseBlackOilModel**

Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.

Added primary variables: \(r_s, r_v\)

---

**ReservoirModel**

The class declares known drive mechanisms:

```matlab
function forces = getValidDrivingForces(model)
forces = getValidDrivingForces @PhysicalModel(model);
forces.W = [];
forces.bc = [];
forces.src = [];
end
```

and define how to evaluate relative permeability, get surface densities, etc.

The class also specifies how to add well equations, source terms, and boundary conditions to the equation system, but does not implement specific flow equations.
**Functionality through inheritance**

<table>
<thead>
<tr>
<th>PhysicalModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract base class for all MRST models. Contains logic related to linearization and updates.</td>
</tr>
<tr>
<td>Primary variables: None</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReservoirModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature. Base class for all reservoir models.</td>
</tr>
<tr>
<td>Added primary variables: $s_\alpha$, $p$, $T$, $q_\alpha$, $p_{bh}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ThreePhaseBlackOilModel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.</td>
</tr>
<tr>
<td>Added primary variables: $r_s$, $r_v$</td>
</tr>
</tbody>
</table>

**ReservoirModel**

Default discretization is a two-point method:

```matlab
function model = ...
    setupOperators(model, G, rock, varargin)
    model.operators = ...
    setupOperatorsTPFA(G, rock, varargin{:});
end
```
Functionality through inheritance

**PhysicalModel**
Abstract base class for all MRST models. Contains logic related to linearization and updates.
Primary variables: None

**ReservoirModel**
Extends PhysicalModel with rock, fluid, saturations, pressures, and temperature. Base class for all reservoir models.
Added primary variables: $s_\alpha, p, T, q_\alpha, p_{bh}$

**ThreePhaseBlackOilModel**
Extends ReservoirModel with optional solution gas and vaporized oil. Base class for two- and single-phase versions.
Added primary variables: $r_s, r_v$

Implementes specific equations, which in this case is a general black-oil model with dissolved gas and vaporized oil.

Evaluation of residual equations:

```matlab
[problem, state] = ... 
equationsBlackOil(state0, state, ...
model, dt, drivingForces, varargin)
```

Details of this function is as given for two-phase case above, but with more features and logic that switches unknowns depending on phases present.
Constructing a simulation model from ECLIPSE input

**Input deck**

- Contains:
  - Input parser
  - Reads complete simulation decks:
    - grid and petrophysics, fluid and rock properties, region information, well definitions, operating schedule, convergence control, etc.

**Data**

- Reservoir model
  - Description of geology and fluid behavior as well as discrete averaging and spatial discretization operators

- Grid
- Petrophysics
- Fluids

**Class**

- State
  - Physical variables inside the reservoir:
    - \( p, s_w, s_o, s_g, c, r_v, r_s \)

- Well state
  - Physical variables inside the wellbore:
    - \( q_w^s, q_s^s, q_g^s, q_p^s, p_{bh} \)

**Struct**

- Schedule
  - Time steps and controls and settings for wells and boundary conditions

- Wells
ECLIPSE input decks

RUNSPEC – simulation description (name of the case, grid dimensions, phases and components present, number of wells, table dimensions, etc)

GRID – grid geometry/topology and petrophysical properties (porosity, permeability, net-to-gross).

EDIT – user-defined changes of pore volume, cell centers, transmissibilities, LGR, etc (optional)

PROPS – rock-fluid and PVT properties

REGIONS – spatial dependence for initialization, rock-fluid and PVT properties (optional)

SOLUTION – specifies how the model is to be initialized

SUMMARY – specifies output of reservoir responses (well curves, average pressure, etc) to summary file after each time step (optional)

SCHEDULE – defines wells and how they are to be operated, time step selection and solver tolerances, controls output of cell properties

Data file: SPE1 benchmark
ECLIPSE input decks

RUNSPEC – simulation description (name of the case, grid dimensions, phases and components present, number of wells, table dimensions, etc)

GRID – grid geometry/topology and petrophysical properties (porosity, permeability, net-to-gross).

EDIT – user-defined changes of pore volume, cell centers, transmissibilities, LGR, etc (optional)

PROPS – rock-fluid and PVT properties

REGIONS – spatial dependence for initialization, rock-fluid and PVT properties (optional)

SOLUTION – specifies how the model is to be initialized

SUMMARY – specifies output of reservoir responses (well curves, average pressure, etc) to summary file after each time step (optional)

SCHEDULE – defines wells and how they are to be operated, time step selection and solver tolerances, controls output of cell properties

Data file: SPE1 benchmark
ECLIPSE input decks

RUNSPEC – simulation description (name of the case, grid dimensions, phases and components present, number of wells, table dimensions, etc)

GRID – grid geometry/topology and petrophysical properties (porosity, permeability, net-to-gross).

EDIT – user-defined changes of pore volume, cell centers, transmissibilities, LGR, etc (optional)

PROPS – rock-fluid and PVT properties

REGIONS – spatial dependence for initialization, rock-fluid and PVT properties (optional)

SOLUTION – specifies how the model is to be initialized

SUMMARY – specifies output of reservoir responses (well curves, average pressure, etc) to summary file after each time step (optional)

SCHEDULE – defines wells and how they are to be operated, time step selection and solver tolerances, controls output of cell properties

---

### PROPS

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relative Permeability</td>
<td>0.100000000000000</td>
</tr>
<tr>
<td>Capillary Pressure</td>
<td>0.997000000000000</td>
</tr>
</tbody>
</table>

---

### SGDF

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Pressure</td>
<td>14.7</td>
</tr>
<tr>
<td>Compressibility</td>
<td>3.0E-6</td>
</tr>
<tr>
<td>Viscosity</td>
<td>0.31</td>
</tr>
</tbody>
</table>

---

### OUTPUT CONTROLS FOR PROPS DATA

<table>
<thead>
<tr>
<th>Activate</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPTPROPS</td>
<td>1</td>
</tr>
<tr>
<td>WFN</td>
<td>1</td>
</tr>
<tr>
<td>SGFN</td>
<td>1</td>
</tr>
<tr>
<td>PVTW</td>
<td>1</td>
</tr>
<tr>
<td>PVDG</td>
<td>1</td>
</tr>
<tr>
<td>DENSITY</td>
<td>1</td>
</tr>
<tr>
<td>ROCK</td>
<td>1</td>
</tr>
</tbody>
</table>

---

### PVT PROPERTIES OF WATER

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Pressure</td>
<td>4.7</td>
</tr>
<tr>
<td>Reference FVF</td>
<td>0.001000000000000</td>
</tr>
<tr>
<td>Compressibility</td>
<td>3.13E-6</td>
</tr>
<tr>
<td>Viscosity</td>
<td>0.31</td>
</tr>
</tbody>
</table>

---

### ROCK COMPRESSIBILITY

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Pressure</td>
<td>14.7</td>
</tr>
<tr>
<td>Compressibility</td>
<td>3.0E-6</td>
</tr>
</tbody>
</table>

---

### SURFACE DENSITIES OF RESERVOIR FLUIDS

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oil Density</td>
<td>49.1</td>
</tr>
<tr>
<td>Water Density</td>
<td>64.79</td>
</tr>
<tr>
<td>Gas Density</td>
<td>0.06054</td>
</tr>
</tbody>
</table>

---

### PVT PROPERTIES OF DRY GAS (NO VAPORISED OIL)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Pressure</td>
<td>14.7</td>
</tr>
<tr>
<td>Reference FVF</td>
<td>0.001000000000000</td>
</tr>
<tr>
<td>Compressibility</td>
<td>3.13E-6</td>
</tr>
<tr>
<td>Viscosity</td>
<td>0.31</td>
</tr>
<tr>
<td>Viscosity</td>
<td>0.31</td>
</tr>
</tbody>
</table>

---

### PVT PROPERTIES OF LIVE OIL (WITH DISSOLVED GAS)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Pressure</td>
<td>1.827</td>
</tr>
<tr>
<td>Reference FVF</td>
<td>4.4900000000e-01</td>
</tr>
<tr>
<td>Compressibility</td>
<td>1.7260000000e+00</td>
</tr>
<tr>
<td>Viscosity</td>
<td>6.0500000000e-01</td>
</tr>
</tbody>
</table>

---

### PVT PROPERTIES OF DRY GAS (30% VAPORIZED OIL)

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Pressure</td>
<td>14.7</td>
</tr>
<tr>
<td>Reference FVF</td>
<td>3.0E-6</td>
</tr>
<tr>
<td>Compressibility</td>
<td>0.31</td>
</tr>
</tbody>
</table>

---

### PVT PROPERTIES OF WET GAS

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference Pressure</td>
<td>14.7</td>
</tr>
<tr>
<td>Reference FVF</td>
<td>3.0E-6</td>
</tr>
<tr>
<td>Compressibility</td>
<td>0.31</td>
</tr>
</tbody>
</table>

---

---

Data file: SPE1 benchmark
ECLIPSE input decks

RUNSPEC – simulation description (name of the case, grid dimensions, phases and components present, number of wells, table dimensions, etc)

GRID – grid geometry/topology and petrophysical properties (porosity, permeability, net-to-gross).

EDIT – user-defined changes of pore volume, cell centers, transmissibilities, LGR, etc (optional)

PROPS – rock-fluid and PVT properties

REGIONS – spatial dependence for initialization, rock-fluid and PVT properties (optional)

SOLUTION – specifies how the model is to be initialized

SUMMARY – specifies output of reservoir responses (well curves, average pressure, etc) to summary file after each time step (optional)

SCHEDULE – defines wells and how they are to be operated, time step selection and solver tolerances, controls output of cell properties

Data file: SPE1 benchmark
ECLIPSE input decks

RUNSPEC – simulation description (name of the case, grid dimensions, phases and components present, number of wells, table dimensions, etc)

GRID – grid geometry/topology and petrophysical properties (porosity, permeability, net-to-gross).

EDIT – user-defined changes of pore volume, cell centers, transmissibilities, LGR, etc (optional)

PROPS – rock-fluid and PVT properties

REGIONS – spatial dependence for initialization, rock-fluid and PVT properties (optional)

SOLUTION – specifies how the model is to be initialized

SUMMARY – specifies output of reservoir responses (well curves, average pressure, etc) to summary file after each time step (optional)

SCHEDULE – defines wells and how they are to be operated, time step selection and solver tolerances, controls output of cell properties

Data file: SPE1 benchmark
ECLIPSE input decks

RUNSPEC – simulation description (name of the case, grid dimensions, phases and components present, number of wells, table dimensions, etc)

GRID – grid geometry/topology and petrophysical properties (porosity, permeability, net-to-gross).

EDIT – user-defined changes of pore volume, cell centers, transmissibilities, LGR, etc (optional)

PROPS – rock-fluid and PVT properties

REGIONS – spatial dependence for initialization, rock-fluid and PVT properties (optional)

SOLUTION – specifies how the model is to be initialized

SUMMARY – specifies output of reservoir responses (well curves, average pressure, etc) to summary file after each time step (optional)

SCHEDULE – defines wells and how they are to be operated, time step selection and solver tolerances, controls output of cell properties

--- SCHEDULE ---

<table>
<thead>
<tr>
<th>Type</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCER</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>INJECTOR</td>
<td>5000</td>
<td>5000</td>
</tr>
</tbody>
</table>

--- YEAR 1 ---

<table>
<thead>
<tr>
<th>Type</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCER</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>INJECTOR</td>
<td>5000</td>
<td>5000</td>
</tr>
</tbody>
</table>

--- YEAR 2 ---

<table>
<thead>
<tr>
<th>Type</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCER</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>INJECTOR</td>
<td>5000</td>
<td>5000</td>
</tr>
</tbody>
</table>

--- YEAR 3 ---

<table>
<thead>
<tr>
<th>Type</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCER</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>INJECTOR</td>
<td>5000</td>
<td>5000</td>
</tr>
</tbody>
</table>

--- 912.50 --> 1000.0 ---

<table>
<thead>
<tr>
<th>Type</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCER</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>INJECTOR</td>
<td>5000</td>
<td>5000</td>
</tr>
</tbody>
</table>

--- 1000.0 --> 1100.0 ---

<table>
<thead>
<tr>
<th>Type</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCER</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>INJECTOR</td>
<td>5000</td>
<td>5000</td>
</tr>
</tbody>
</table>

--- 1100.0 --> 1200.0 ---

<table>
<thead>
<tr>
<th>Type</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRODUCER</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>INJECTOR</td>
<td>5000</td>
<td>5000</td>
</tr>
</tbody>
</table>

--- SUMMARY ---

Data file: SPE1 benchmark
Example: the SPE 9 benchmark

- Grid: $24 \times 25 \times 15$, 9000 cells
- 3-phase model, dissolved gas but no vaporized oil
- 1 water injector, rate controlled, switches to bhp
- 25 producers, oil-rate controlled, most switch to bhp
- Appearance of free gas due to pressure drop

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

Example: the SPE 9 benchmark

- Grid: $24 \times 25 \times 15$, 9000 cells
- 3-phase model, dissolved gas but no vaporized oil
- 1 water injector, rate controlled, switches to bhp
- 25 producers, oil-rate controlled, most switch to bhp
- Appearance of free gas due to pressure drop

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

Reading input and construct basic MRST objects:

```matlab
pth = fullfile(getDatasetPath('spe9'), 'BENCH_SPE9.DATA');
deck = readEclipseDeck(fn);
deck = convertDeckUnits(deck);

G = initEclipseGrid(deck);
G = computeGeometry(G);

rock = initEclipseRock(deck);
rock = compressRock(rock, G.cells.indexMap);

fluid = initDeckADIFluid(deck);
```

Example: the SPE 9 benchmark

- Grid: $24 \times 25 \times 15$, 9000 cells
- 3-phase model, dissolved gas but no vaporized oil
- 1 water injector, rate controlled, switches to bhp
- 25 producers, oil-rate controlled, most switch to bhp
- Appearance of free gas due to pressure drop

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

Initialization from given state in the input file:

```matlab
gravity reset on
p0 = deck.SOLUTION.PRESSURE;
sw0 = deck.SOLUTION.SWAT;
sg0 = deck.SOLUTION.SGAS;
s0 = [sw0, 1-sw0-sg0, sg0];
rs0 = deck.SOLUTION.RS;

state = struct('s', s0, 'rs', rs0, 'rv', rv0, 'pressure', p0);
```

Generally, one may have to solve an equilibrium problem to set the initial state.

Example: the SPE 9 benchmark

- Grid: $24 \times 25 \times 15$, 9000 cells
- 3-phase model, dissolved gas but no vaporized oil
- 1 water injector, rate controlled, switches to bhp
- 25 producers, oil-rate controlled, most switch to bhp
- Appearance of free gas due to pressure drop

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

Create model and simulation schedule:

```matlab
model = selectModelFromDeck(G, rock, fluid, deck);

% Set maximum limits on changes in saturation, Rs and pressure
model.drsMaxRel = .2;
model.dpMaxRel = .2;
model.dsMaxAbs = .05;

% Convert the deck schedule into a MRST schedule
schedule = convertDeckScheduleToMRST(model, deck);
```

Example: the SPE 9 benchmark

- Grid: $24 \times 25 \times 15$, 9000 cells
- 3-phase model, dissolved gas but no vaporized oil
- 1 water injector, rate controlled, switches to bhp
- 25 producers, oil-rate controlled, most switch to bhp
- Appearance of free gas due to pressure drop

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

Select linear solver:

```try
mrstModule add agmg
pressureSolver = AGMGSolverAD('tolerance', 1e-4);
catch
pressureSolver = BackslashSolverAD();
end
linsolve = CPRSolverAD('ellipticSolver', pressureSolver);
```

We select a CPR-type solver, with AGMG as multigrid preconditioner. The CPR preconditioner attempts to decouple the linear system into a pressure component and a transport component. Although not necessary here, it improves CPU time.

Example: the SPE 9 benchmark

- Grid: $24 \times 25 \times 15$, 9000 cells
- 3-phase model, dissolved gas but no vaporized oil
- 1 water injector, rate controlled, switches to bhp
- 25 producers, oil-rate controlled, most switch to bhp
- Appearance of free gas due to pressure drop

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

Inspect the rock-fluid and PVD properties:

```python
inspectFluidModel(model)
```

The AD-OO framework can interactively visualize the fluid model of a ReservoirModel instance. Once active, the user can interactively explore the different fluid properties (viscosities, relative permeabilities, densities) as functions of saturation and pressure.
Example: the SPE 9 benchmark

- Grid: 24 × 25 × 15, 9000 cells
- 3-phase model, dissolved gas but no vaporized oil
- 1 water injector, rate controlled, switches to bhp
- 25 producers, oil-rate controlled, most switch to bhp
- Appearance of free gas due to pressure drop

Run the schedule

```python
model.verbose = True;
fn = getPlotAfterStep(state0, model, schedule, ...
    'plotWell', False, 'plotReservoir', False);
[wellsols, states, reports] = ...
    simulateScheduleAD(state0, model, schedule, ...
        'LinearSolver', linsolve, 'afterStepFn', fn);
```

We give the schedule with well controls and control time steps. The simulator may use other timesteps internally, but it will always return values at the specified control steps. Setting `model.verbose=false` removes extensive reports about convergence, etc.

---

Example: the SPE 9 benchmark

- Grid: \(24 \times 25 \times 15\), 9000 cells
- 3-phase model, dissolved gas but no vaporized oil
- 1 water injector, rate controlled, switches to bhp
- 25 producers, oil-rate controlled, most switch to bhp
- Appearance of free gas due to pressure drop

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

Launch a viewer to inspect reservoir responses:

```
plotWellSols(wellsols, cumsum(schedule.step.val), 'field', 'qTr')
```

Here, you can plot bottom-hole pressures, reservoir and surface rates, oil and water cut, gas-oil ratio, etc. Plots are versus time or time step, and can be instantaneous or cummulative.
Example: the SPE 9 benchmark

- Grid: $24 \times 25 \times 15$, 9000 cells
- 3-phase model, dissolved gas but no vaporized oil
- 1 water injector, rate controlled, switches to bhp
- 25 producers, oil-rate controlled, most switch to bhp
- Appearance of free gas due to pressure drop

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

MRST also offers functionality for processing ECLIPSE output. We can use this to compare results from the two simulators:

```matlab
compare = fullfile(mrstPath('ad-blackoil'), 'examples', 'spe9', 'compare');
smry = readEclipseSummaryUnFmt(fullfile(compare, 'SPE9'));
compd = 1:(size(smry.data, 2));
Tcomp = smry.get(':+:+:+:+', 'YEARS', compd);
comp = convertFrom(smry.get('PROD13', 'WBHP', compd), psia);

T = convertTo(cumsum(schedule.step.val), year);
mrst = getWellOutput(wellsols, 'bhp', 'PROD13');

plot(T, mrst, Tcomp, comp);
```

Go through some of the tutorials from the ad-blackoil module. In particular, I recommend:

- `spe9/blackoilTutorialSPE9` – we only covered parts of it here
- `simulatorWorkflowExample` – a complete example that does not use ECLIPSE input
- `multisegmentWellExample` – shows use of multisegment well models
- `blackoilSectorModelExample` – specification of boundary conditions