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

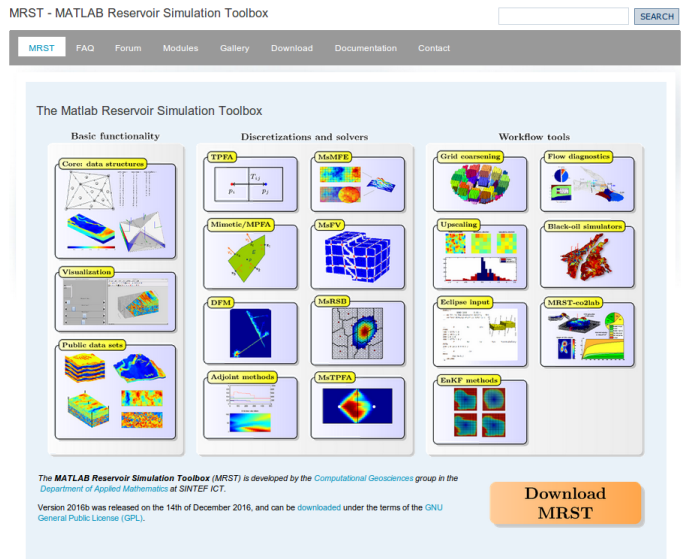
Outline

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



`http://www.sintef.no/MRST`

Used in publications:

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

Toolbox for experimental programming

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

Diagnostics/proxies

- Flow diagnostics/volumetrics
- Physics-based proxies
- Fast optimization
- Spill-point analysis

Model reduction

- Grid coarsening
- Flow-based upscaling
- Multiscale methods
- Model-reduction techniques
- Vertical-equilibrium models

Full simulation

- Black-oil, EOR, thermal, compositional, geomechanics
- Grids and discretizations
- Nonlinear/linear solvers
- Rapid prototyping
- Adjoint formulations and (closed-loop) optimization

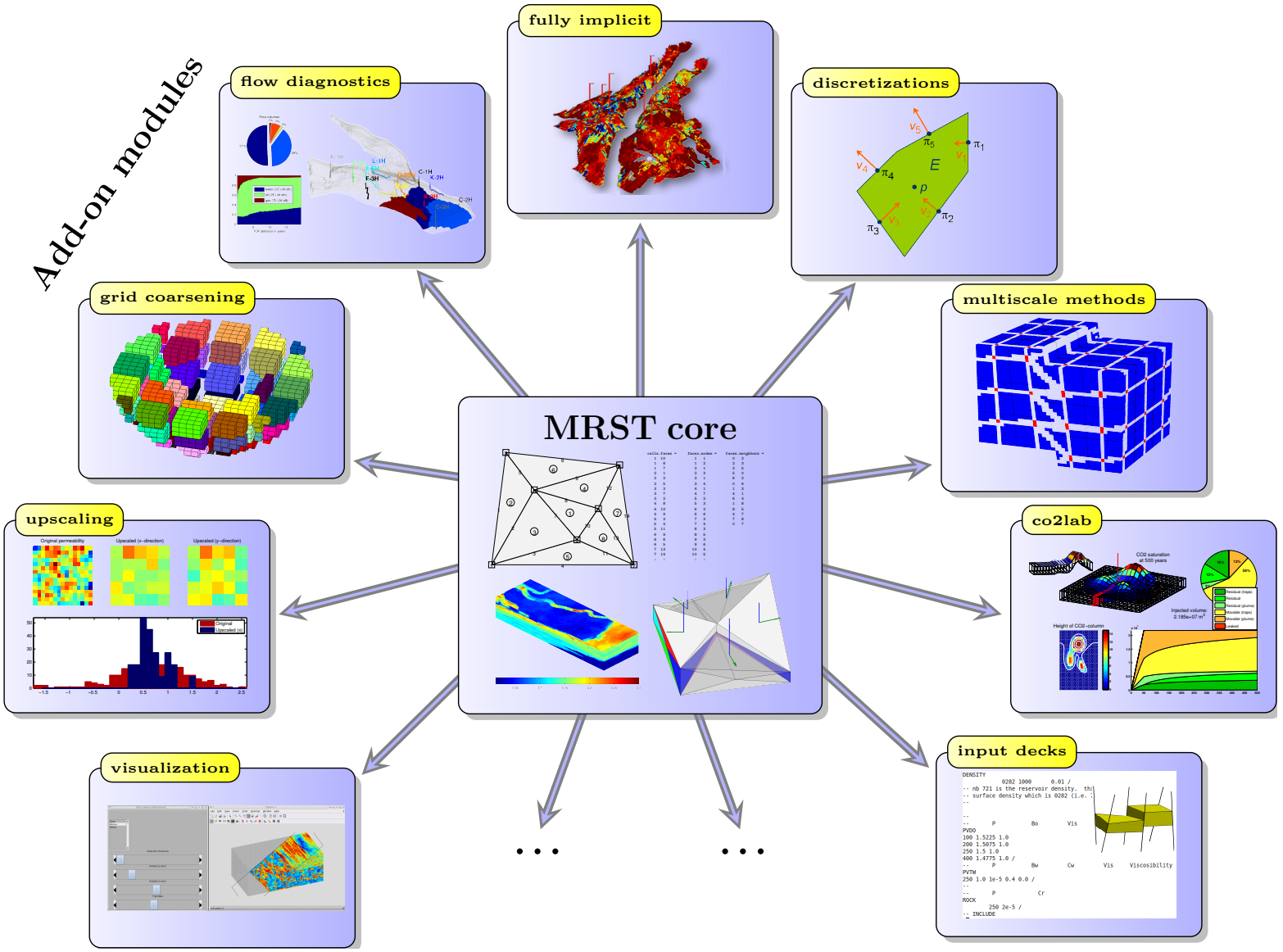
Why in MATLAB?

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 . . .



. . . core and add-on modules

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 CO_2 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

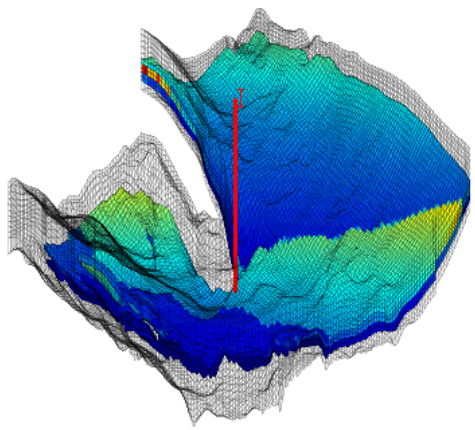
- More recent addition to MRST
- Object-oriented framework for building simulators
- Assembly and linearization performed implicitly by the use of automatic differentiation

Both families rely on functionality from `mrst-core`

Quick overview of functionality

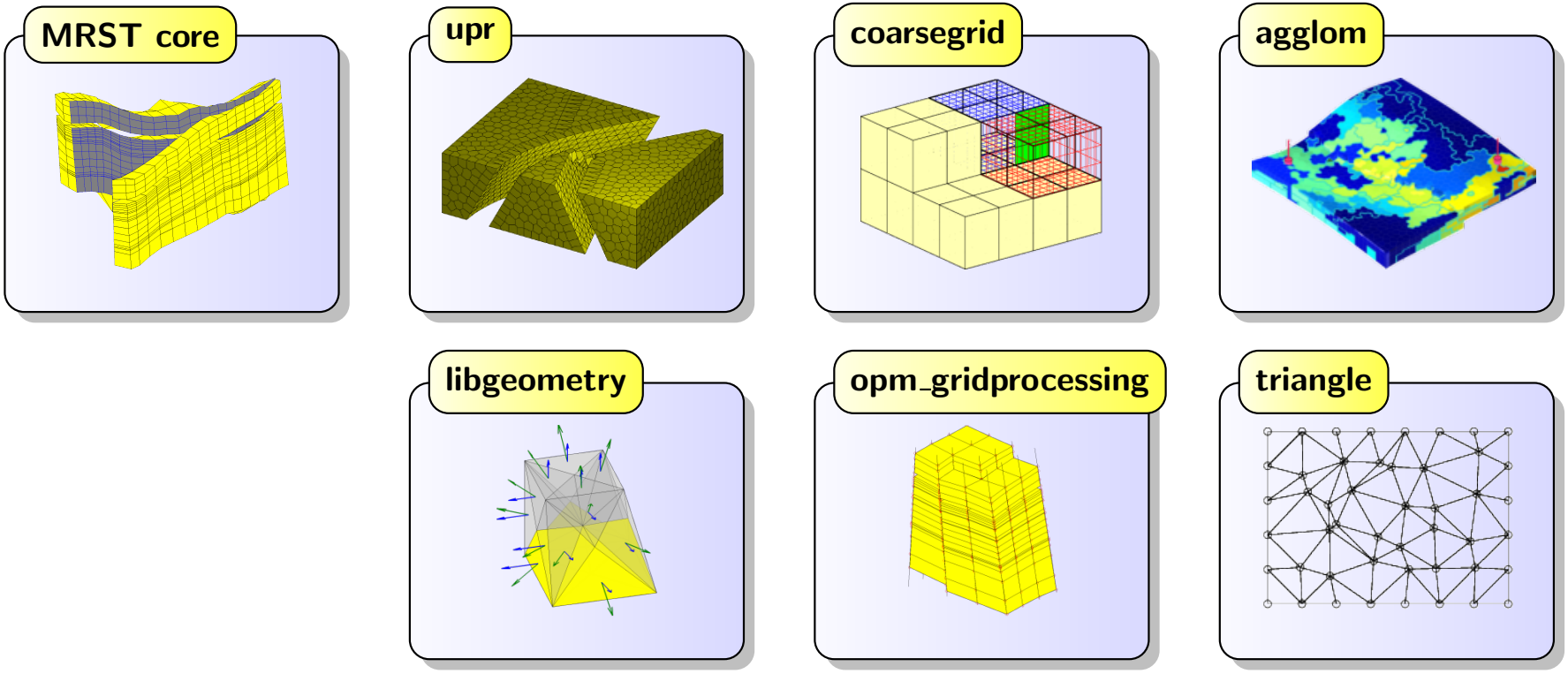
Basic functionality in MRST:

- 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:



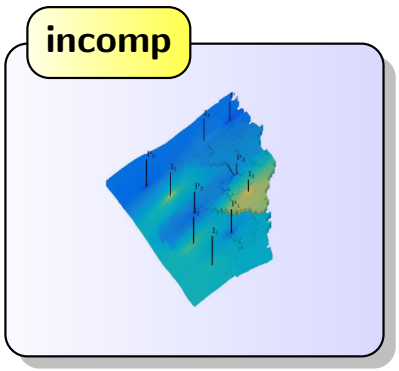
MRST offers a wide variety of grid factory routines and input from Eclipse, `upr` generates 2D and 3D Voronoi grids with cell and face constraints

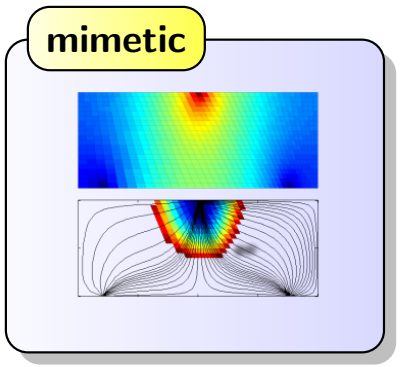
`coarsegrid`: data structures and simple coarsening, adapted partitions in `agglom`

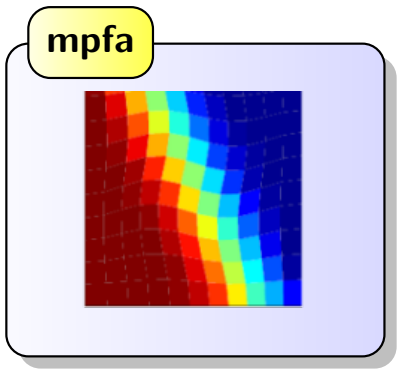
C-accelerated: processing in `libgeometry` and `opm_processing`.

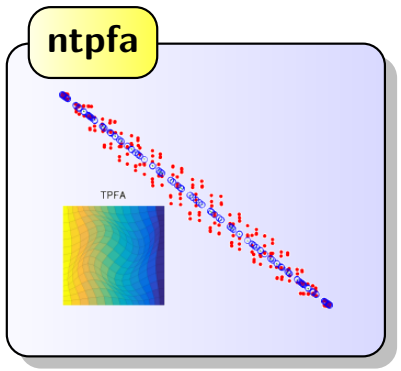
Quick overview of functionality

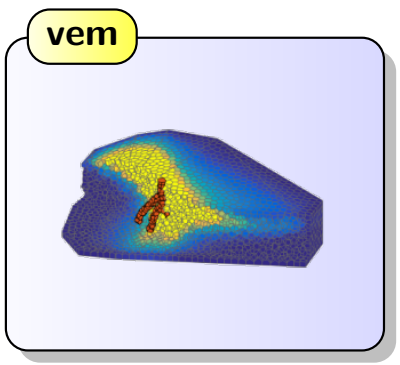
Discretization and solvers for incompressible flow:

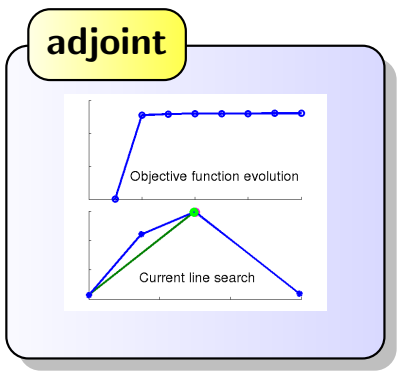










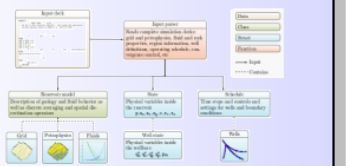


`incomp` implements TPFA-based flow solver and explicit/implicit transport solvers; `mimetic`, `mpfa`, `ntpfa`, and `vem` implement consistent discretizations.

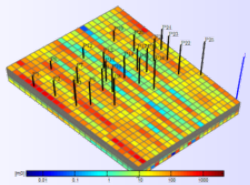
Quick overview of functionality

Discretization and solvers for compressible flow:

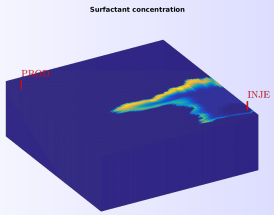
ad-core



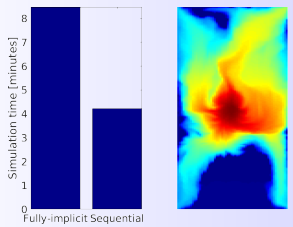
ad-blackoil



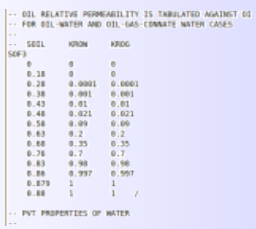
ad-eor



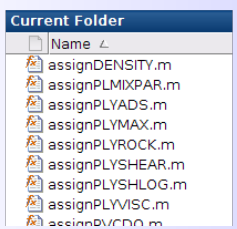
blackoil-sequential



deckreader



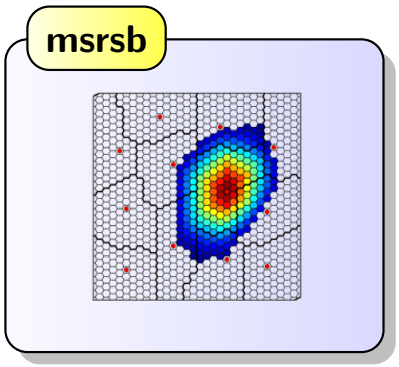
ad-props

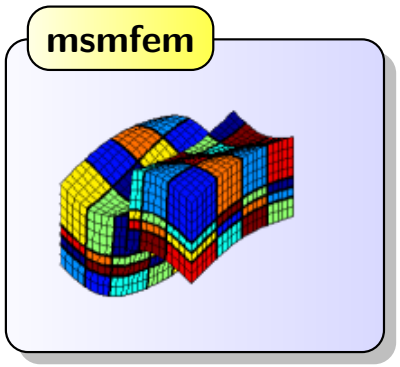


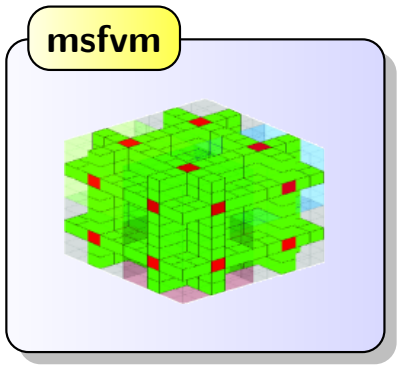
The AD-OO framework offers fully implicit simulators from industry-standard input decks, including computations of adjoints

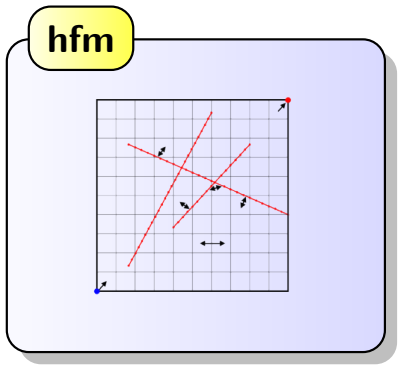
Quick overview of functionality

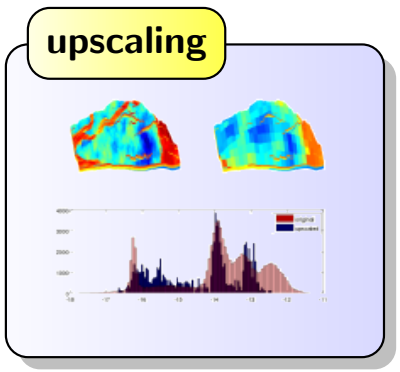
Upscaling and multiscale methods:

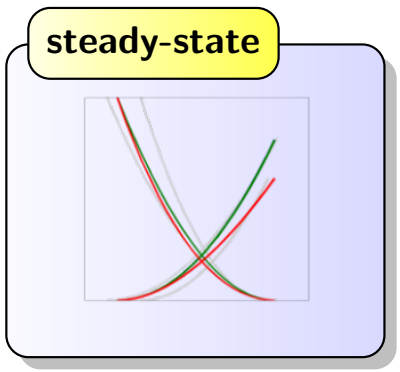










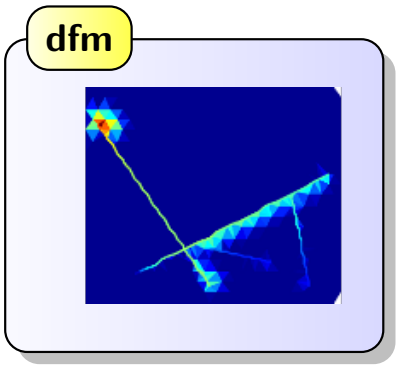


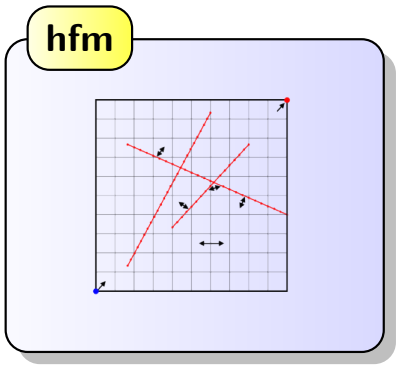
`msrsb` is state-of-the art multiscale solver, `hfm` implements this for fracture models, `msmfem` and `msfvm` are earlier developments

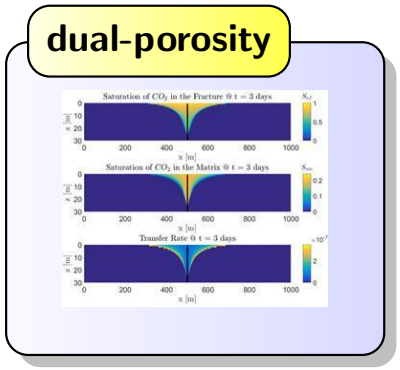
`upscaling`: flow-based single-phase upscaling, `steady-state`: multiphase upscaling

Quick overview of functionality

Fractured media:

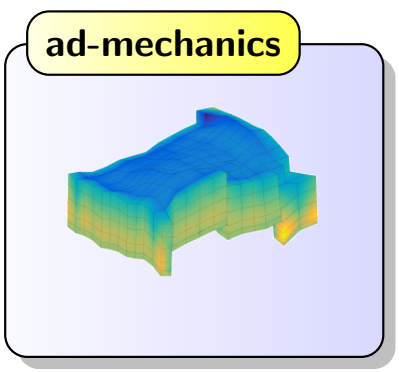


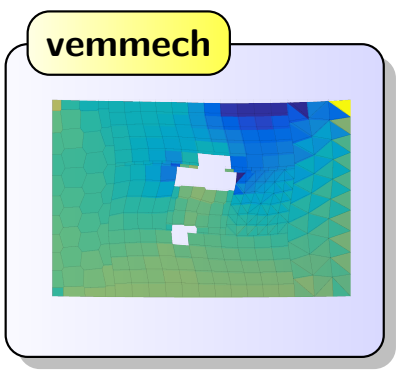


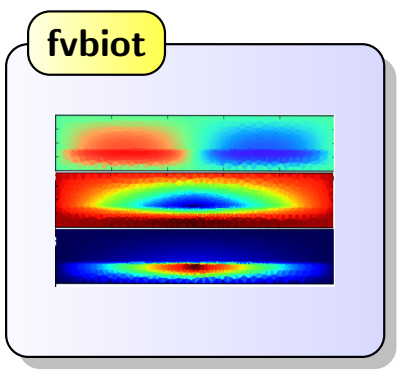


`dfm` – discrete fracture models, `hfm` – hierarchical/embedded fracture models

Geomechanics:



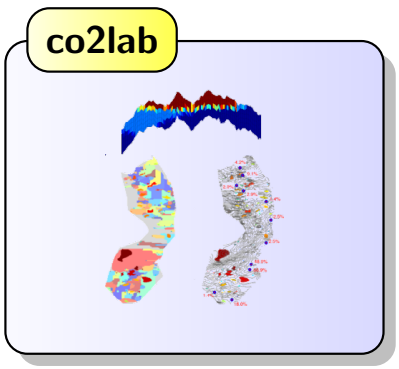


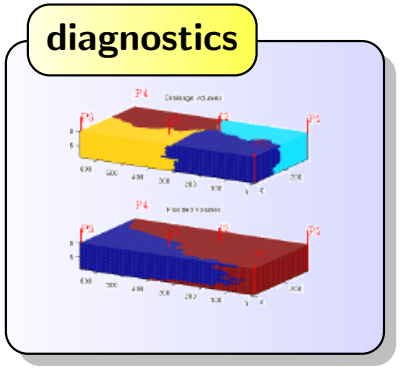


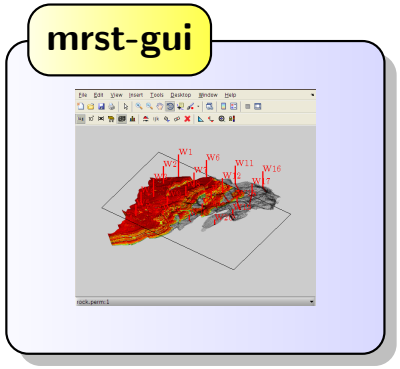
`ad-mechanics`* – coupled flow and mechanics (R2017b), `vemmech` – virtual element methods, `fvbiot` – multipoint stress-approximation methods

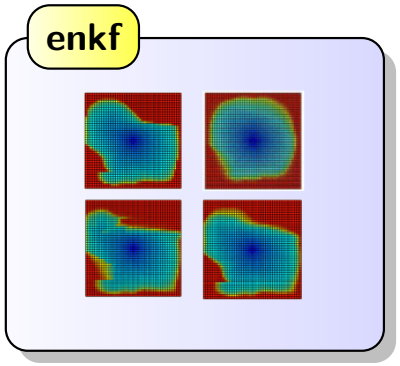
Quick overview of functionality

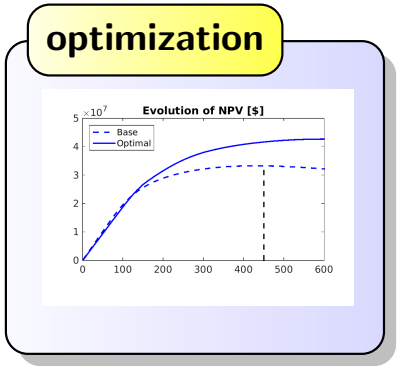
Workflow tools:

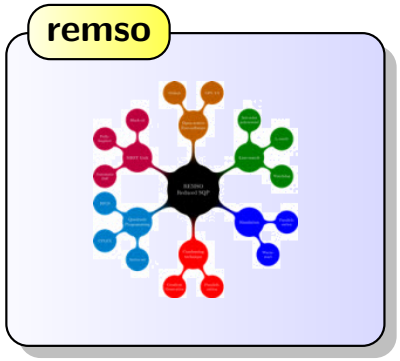












`co2lab`: comprehensive tools for large-scale CO_2 storage saline aquifers

`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

Downloading the software

MRST - MATLAB Reservoir Simulation Toolbox

SEARCH

MRST FAQ Forum Modules Gallery Download Documentation Contact

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, visalisation, 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 CO_2 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`

Installing 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\
```

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

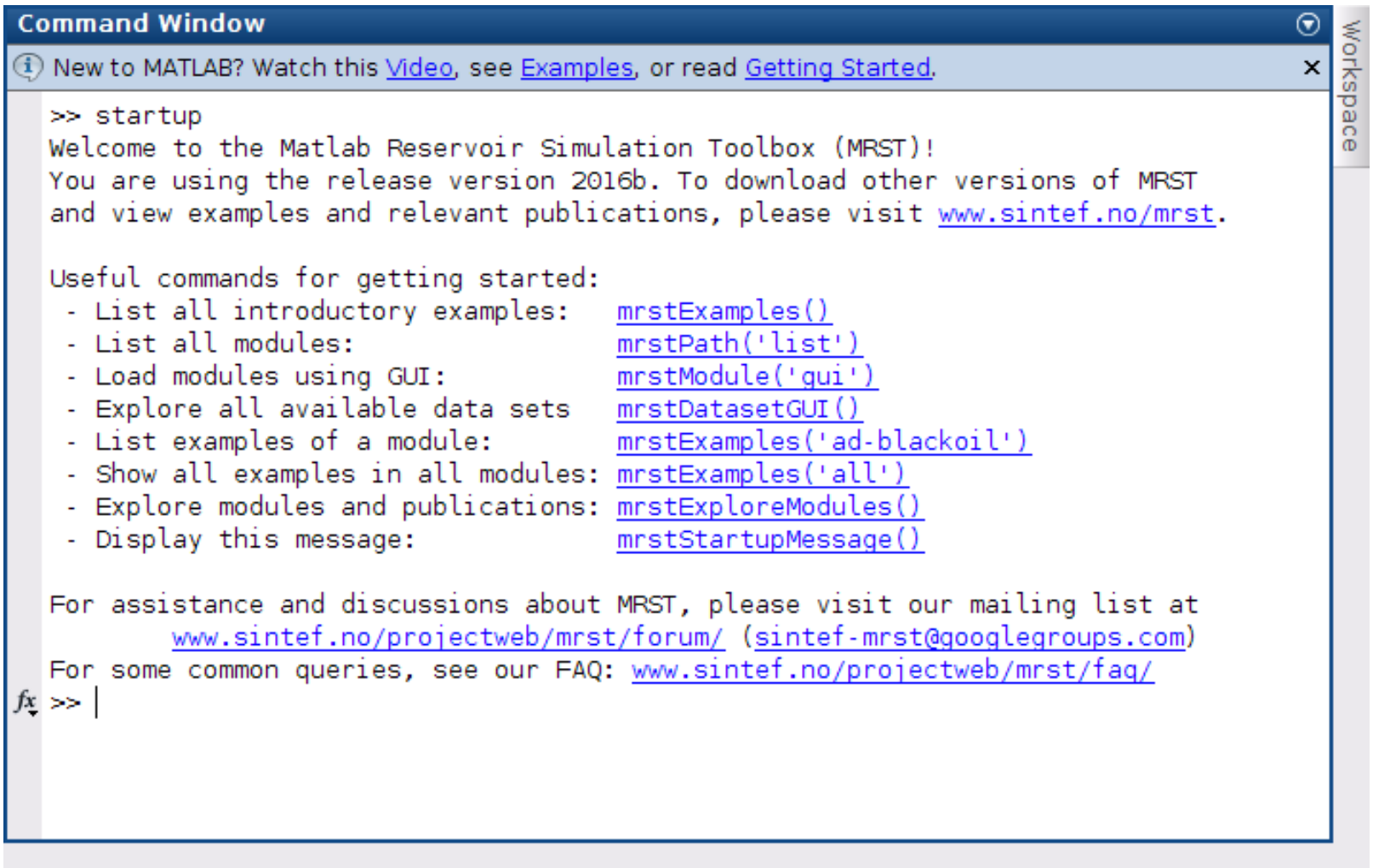
```
startup;
```

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



Getting started: sources for information

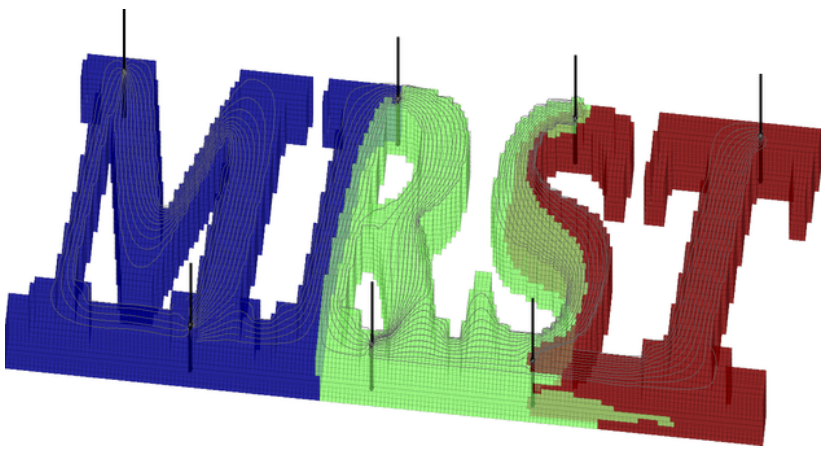
- the MRST book and key publications – the book gives a comprehensive introduction to basic flow simulation as implemented in MRST; three papers give more condensed overviews
- example scripts
- Jolts (just-in-time online learning tools)
- module examples
- manual pages for individual routines
- the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

Knut-Andreas Lie

An Introduction to Reservoir Simulation Using MATLAB

User Guide for the Matlab Reservoir Simulation Toolbox (MRST)

June 12, 2017



SINTEF ICT, Departement of Applied Mathematics
Oslo, Norway

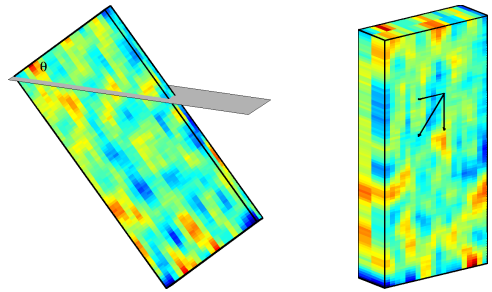


Fig. 10.13. Illustration of the sloping sandbox used for the buoyancy example and how it is simulated by rotating the gravity vector. (Color: Gaussian porosity field).

```
R = makehgtform('yrotate',-pi*theta/180);
gravity reset on
gravity( 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,

```
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,

```
dT = [.5, .5, 1, 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.

Example scripts and tutorials . . .

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

Editor - /home/kalie/jolt/mrst-2014b/examples/1ph/simpleBC.m

File Edit Text Go Cell Tools Debug Desktop Window Help

```
%% Process geometry
% Having set up the basic structure, we continue to compute centroids and
% volumes of the cells and centroids, normals, and areas for the faces. For
% a Cartesian grid, this information can trivially be computed, but is
% given explicitly so that the flow solver is compatible with fully
% unstructured grids.
G = computeGeometry(G);


%% Set rock and fluid data
% The only parameters in the single-phase pressure equation are the
% permeability $K$ and the fluid viscosity $\mu$. We set the permeability
% to be homogeneous and anisotropic
%
% $$ K = \left(\begin{array}{ccc}
%      1000 & 0  & 0 \\ 0 & 100 & 0 \\ 0 & 0 & 10 \end{array}\right) $$
%
% The viscosity is specified by saying that the reservoir is filled with a
% <matlab:help('initSinglefluid') single fluid>, for which de default
% viscosity value equals unity. Our flow solver is written for a general
% incompressible flow and requires the evaluation of a total mobility,
% which is provided by the <matlab:help('fluid') fluid object>.
gravity off
rock.perm = repmat([1000, 100, 10].* milli*darcy(), [G.cells.num, 1]);
fluid     = initSingleFluid('mu' ,    1*centi*poise     , ...
                            'rho', 1014*kilogram/meter^3);

%% Initialize reservoir simulator
% To simplify communication among different flow and transport solvers, all
% unknowns are collected in a structure. Here this structure is initialized
% with uniform initial reservoir pressure equal 0 and (single-phase)
% saturation equal 0.0 (using the default behavior of
% <matlab:help('initResSol') initResSol>)
resSol = initResSol(G, 0.0);
display(resSol);

%% Impose Dirichlet boundary conditions
% Our flow solvers automatically assume no-flow conditions on all outer
% (and inner) boundaries; other type of boundary conditions need to be
% specified explicitly.
%
% Here, we impose Neumann conditions (flux of 1 m^3/day) on the global
% left-hand side. The fluxes must be given in units of m^3/s, and thus we
% need to divide by the number of seconds in a day (<matlab:help('day')
% day()>).  Similarly, we set Dirichlet boundary conditions p = 0 on the
% global right-hand side of the grid, respectively. For a single-phase
% flow, we need not specify the saturation at inflow boundaries. Similarly,
% fluid composition over outflow faces (here, right) is ignored by pside.
bc = fluxside([], G, 'LEFT',  1*meter^3/day());
```

Basic Flow-Solver Tutorial

File Edit View Go Debug Desktop Window Help

Location: /home/kalie/jolt/mrst-2014b/examples/1ph/html/simpleBC.html

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

$$\nabla \cdot v = q, \qquad v = -\frac{K}{\mu}\nabla p,$$

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

Construct a Cartesian grid of size 10-by-10-by-4 cells, where each cell has dimension 1-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; ny = 10; nz = 4;
G = cartGrid([nx, ny, nz]);
display(G);
```

```
G =

        cells: [1x1 struct]
        faces: [1x1 struct]
```

Just-in-time online learning tools

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

Module examples on the web

MRST - MATLAB Reservoir Simulation Toolbox

SEARCH

MRST FAQ Forum Modules Gallery Download Documentation Contact

You are here: MRST / Modules / **Grid Coarsening**

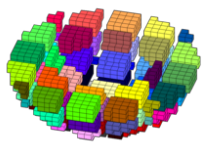
Grid Coarsening

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

Tutorials

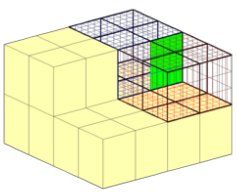
Example 1

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.



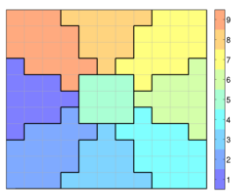
Example 2

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 quater-circle cutout, and a corner-point grid with a single fault.



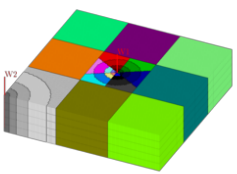
Example 3

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.



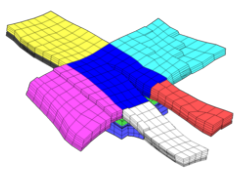
Example 4

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

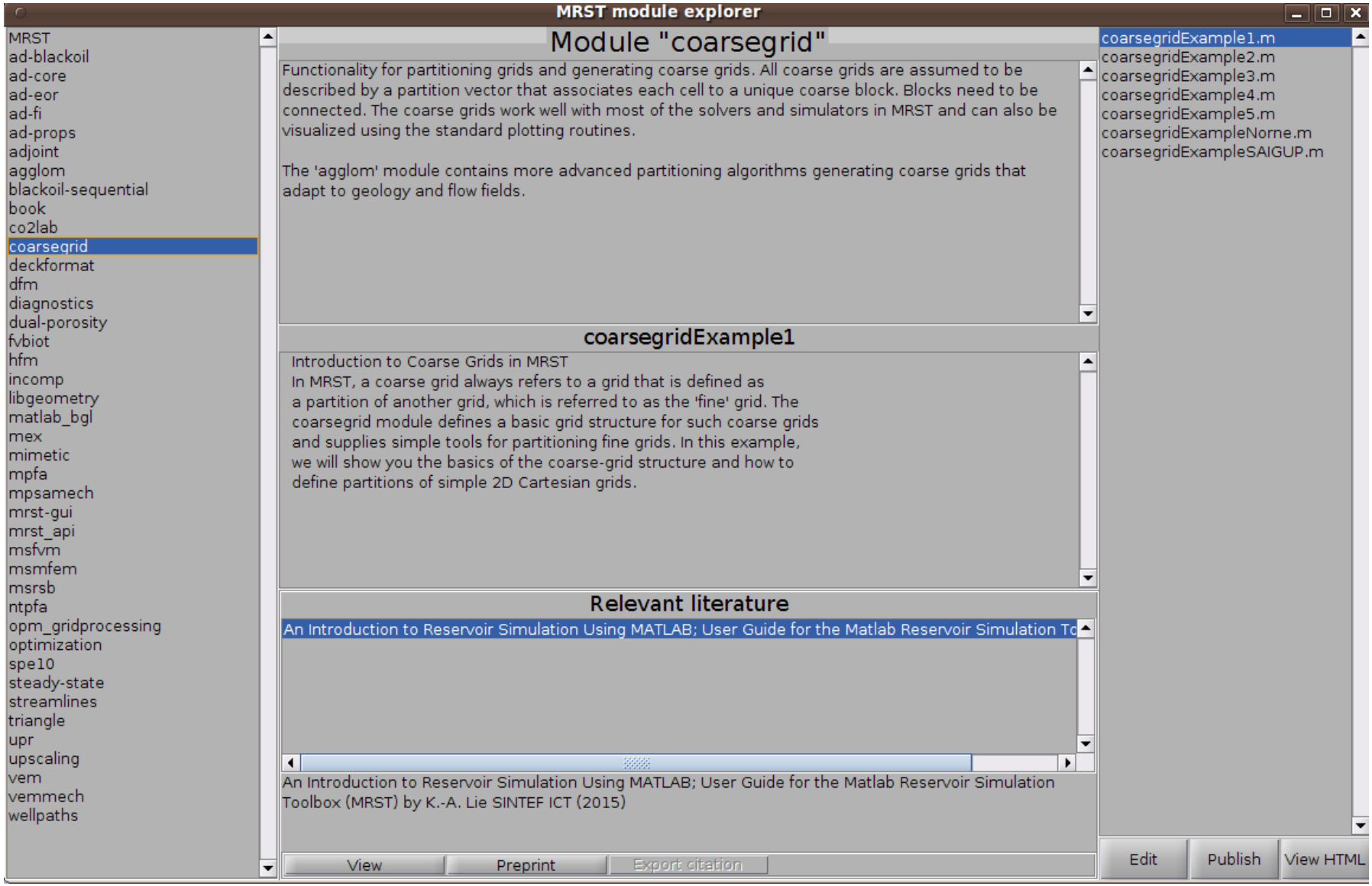
In this example, we use the function `refineNearWell` to make coarse grids with various types of near-well refinement. The examples uses both Cartesian and 2.5 D PEBI fine grids.



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 contigous partition vector. We visualize some of the coarse blocks and show how they are connected with their neighbors.

The module explorer



Start GUI with the command: mrstExploreModules

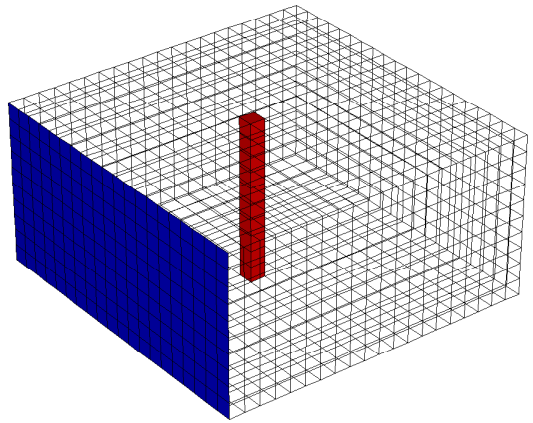
Example: tutorial from `incomp` module

$$\nabla \cdot \vec{v} = q, \qquad \vec{v} = -\frac{\mathbf{K}}{\mu}\left[\nabla p + \rho g \nabla z\right]$$

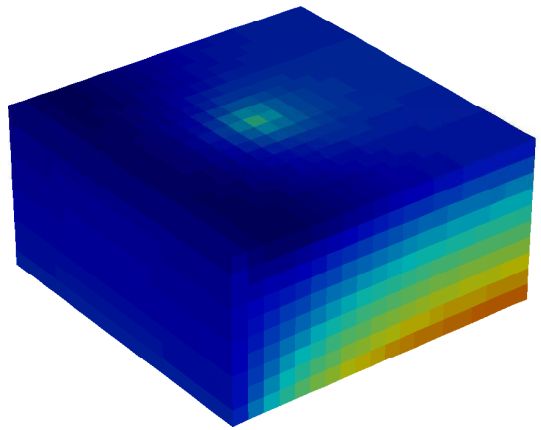
Vertical well and Dirichlet boundary

```
% 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,T,fluid,'src',src,'bc',bc);
p = convertTo(rSol.pressure, barsa());
```

From tutorial: `incompTutorialSRCandBC.m`



Source term and boundary condition

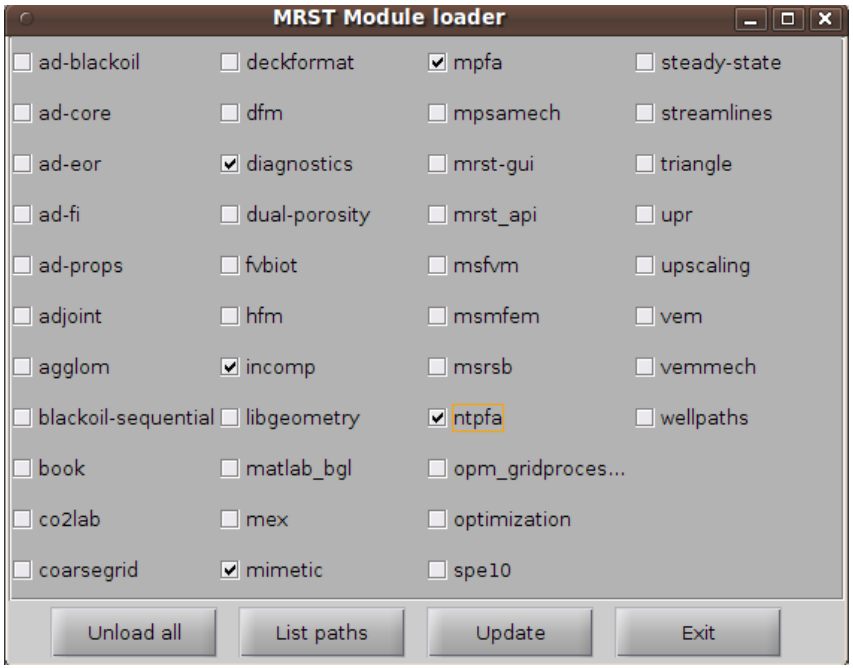


Pressure distribution

Operating modules

Graphical user interface to modules:

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



List all modules and their path

```
mrstPath
```

Load new modules

```
mrstModule add mimetic mpfa
```

Adding your own modules

```
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

Manual pages: computing transmissibility

```
>> help computeTrans
 Compute transmissibilities.

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

  PARAMETERS:
    G    - Grid structure as described by grid_structure.

    rock - Rock data structure with valid field 'perm'.  The permeability
           is assumed to be in measured in units of metres squared (m^2).
           Use function 'darcy' to convert from darcies to m^2, e.g.,

                  perm = convertFrom(perm, milli*darcy)

           if the permeability is provided in units of millidarcies.

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

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

Source code: computing transmissibility

```
% 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.\n', ...
            '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;
   T = T./ sum(C.*C,2);
   clear C;
```

From computeTrans.m

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:

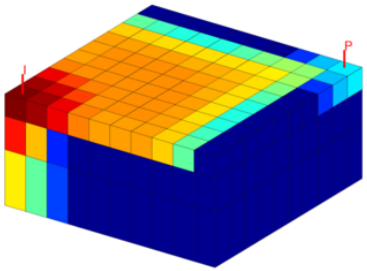
- Search the forum to check if your question has been answered
- Formulate your question carefully. If we cannot understand, we cannot help you
- For code-technical issues, prepare a complete minimal example
- Please help out in answering questions from other users
- Have a little patience; this is not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

URL: `https://groups.google.com/forum/#!forum/sintef-mrst`

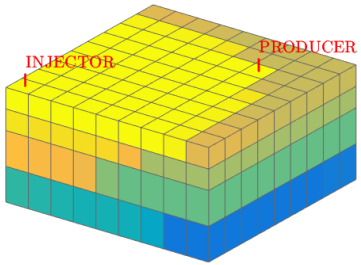
Finding more information . . .

- 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

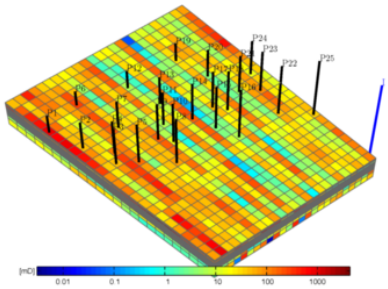
Public data sets



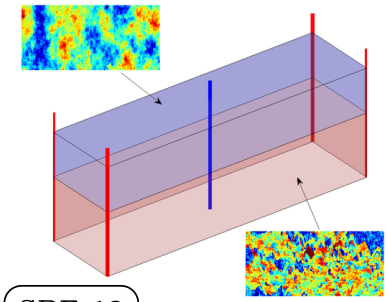
SPE 1



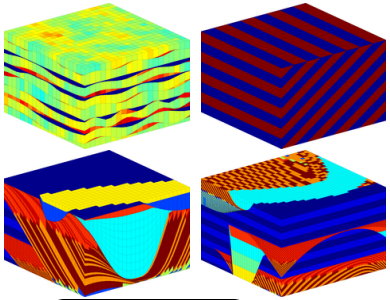
SPE 3



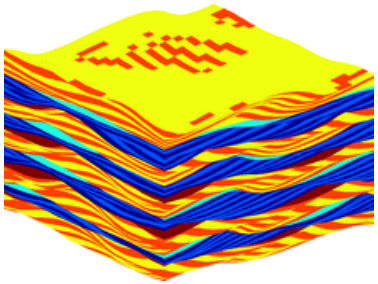
SPE 9



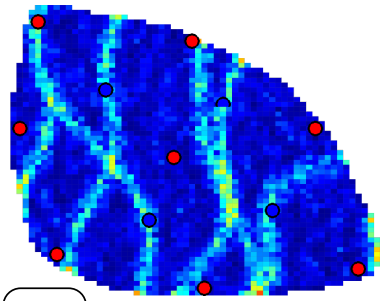
SPE 10



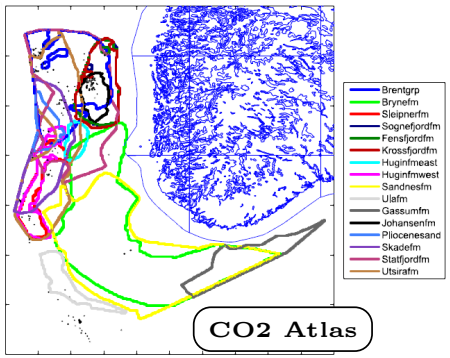
BedModels1



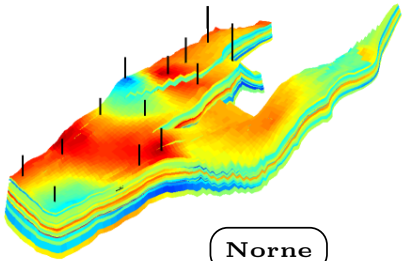
BedModel2



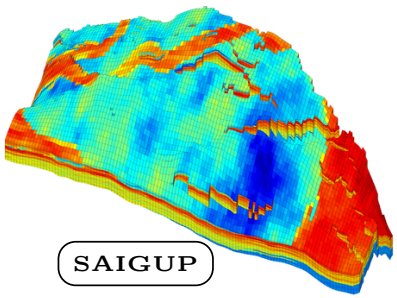
Egg



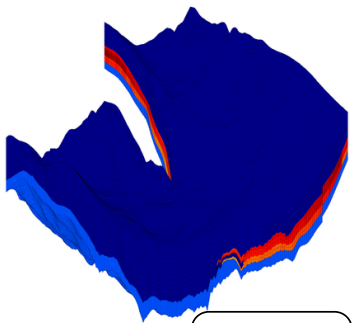
CO2 Atlas



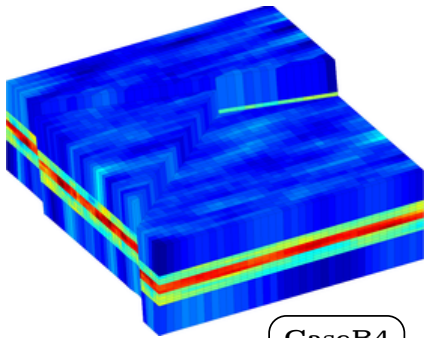
Norne



SAIGUP



Johansen

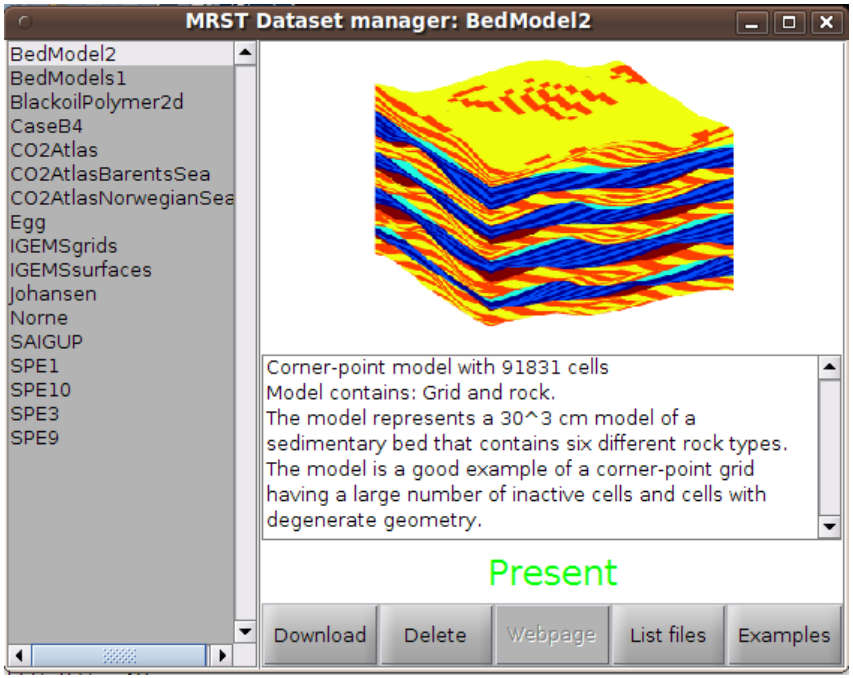


CaseB4

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, `[~,i]=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/`

External dependencies

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/`

Terms of use

You are free to use the software within the GNU GPL3 license, but . . .

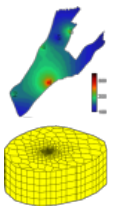
Citing MRST

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):

K. Bao, K.-A. Lie, O. Møyner, and M. Liu. Fully implicit simulation of polymer flooding with MRST. Comput. Geosci., 2017. DOI: 10.1007/s10596-017-9627-2. Also available from: Springer Nature ShareIt.

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

S. Krogstad, K.-A. Lie, O. Møyner, H. M. Nilsen, X. Raynaud, and B. Skaflestad. MRST-AD - an open-source framework for rapid prototyping and evaluation of reservoir simulation problems. Paper 173317-MS presented at the 2015 Reservoir simulation Symposium, Houston, Texas, USA, 23-25 February 2015.



K.-A. Lie, S. Krogstad, I. S. Ligaarden, J. R. Natvig, H. M. Nilsen, and B. Skaflestad. Open source MATLAB implementation of consistent discretisations on complex grids. *Comput. Geosci.,* , Vol. 16, No. 2, pp. 297-322, 2012. DOI: 10.1007/s10596-011-9244-4
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.

An earlier version was published as: K.-A. Lie, S. Krogstad, I. S. Ligaarden, J. R. Natvig, H. M. Nilsen, and B. Skaflestad. Discretisation on complex grids -- Open source MATLAB implementation. Proceedings of ECMOR XII, Oxford, UK, 6-9 September 2010

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 illustrateive 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

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 tast of plotting routines, common tricks, etc

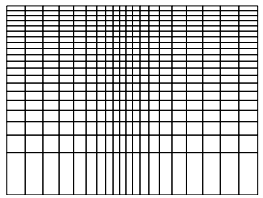
To learn more:

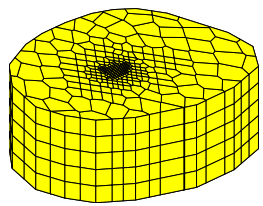
- watch the videos in Jolt2
- study tutorials called `gridTutorial*.m` in `mrst-core`
- read Lie et al. (COMG, 2012), doi: 10.1007/s10596-011-9244-4
- read Chapters 2 and 3 in the MRST book

Grids and physical quantities

The fundamental object in MRST is the grid:

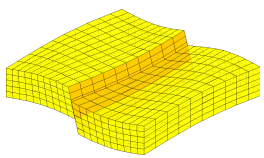
- all grids are assumed to be unstructured
- data structure for geometry and topology
- several grid factory routines
- input of industry-standard formats

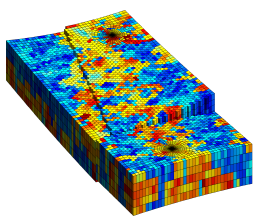




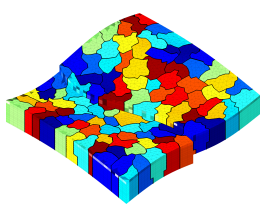
Physical quantities defined as dynamic objects in Matlab:

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





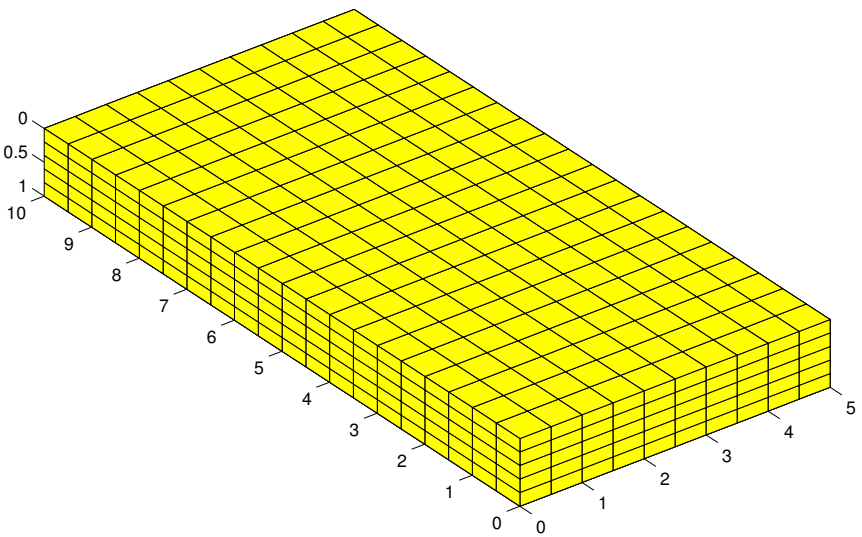
Functions in MRST accept these objects as input, manipulate them, and produce them as output



Standard grids: Cartesian grids

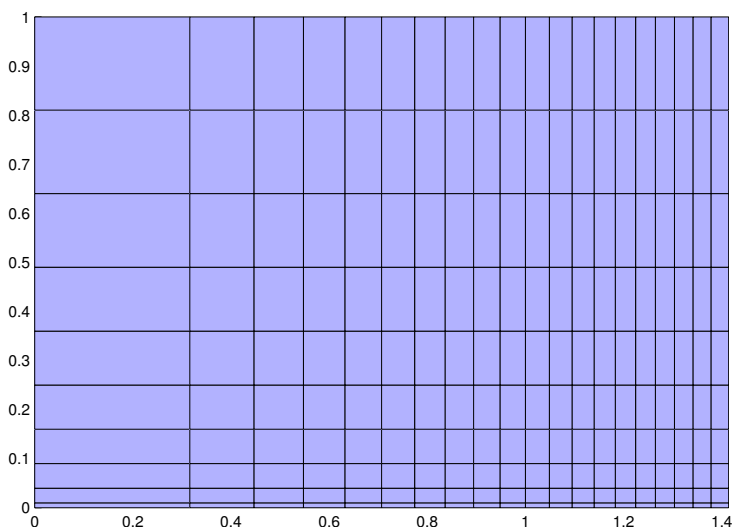
Regular Cartesian grids:

```
G = cartGrid([nx, ny], [Lx Ly]);
G = cartGrid([nx, ny, nz], [Lx Ly Lz]);

G = cartGrid([10 20 5],[5 10 1]);
plotGrid(G), view(3); axis equal
```



Rectilinear grids:

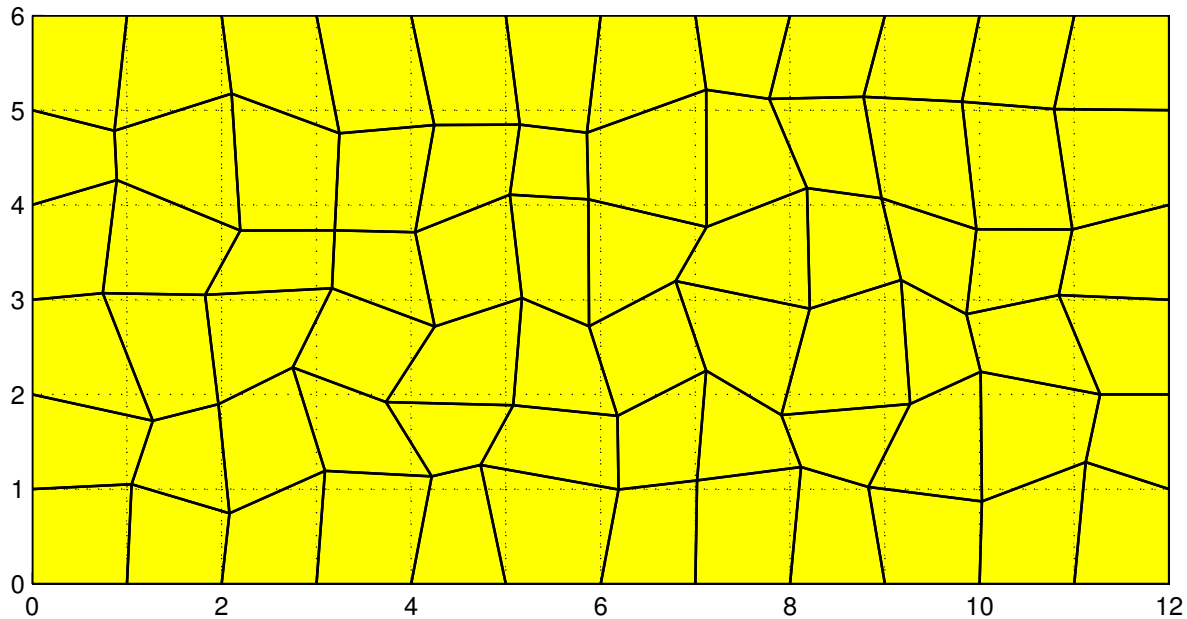
```
G = tensorGrid(x, y);
G = tensorGrid(x, y, z);

G = tensorGrid(sqrt(0:.1:2),(0:.1:1).^2);
plotGrid(G,'FaceColor',[.7 .7 1]);
```



Standard grids: curvilinear grids

Create a rough grid by perturbing the inner nodes randomly

```
nx = 12; ny=6;
G = cartGrid([nx, ny]);
plotGrid(G,'LineStyle',':');

I = true(G.nodes.num,1);          % Logical indexing is fast in MATLAB
I( gridFaceNodes(G, boundaryFaces(G))) = false;
G.nodes.coords(I,:) = G.nodes.coords(I,:) + .6*(rand(sum(I),2)-.5);
plotGrid(G,'FaceColor','none','LineWidth',1);
```

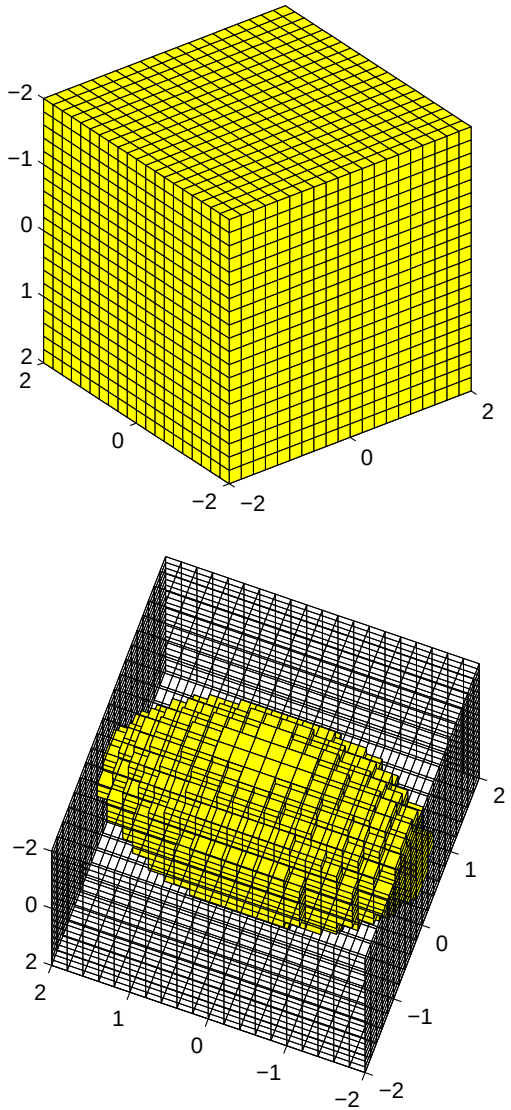


Standard grids: fictitious domains

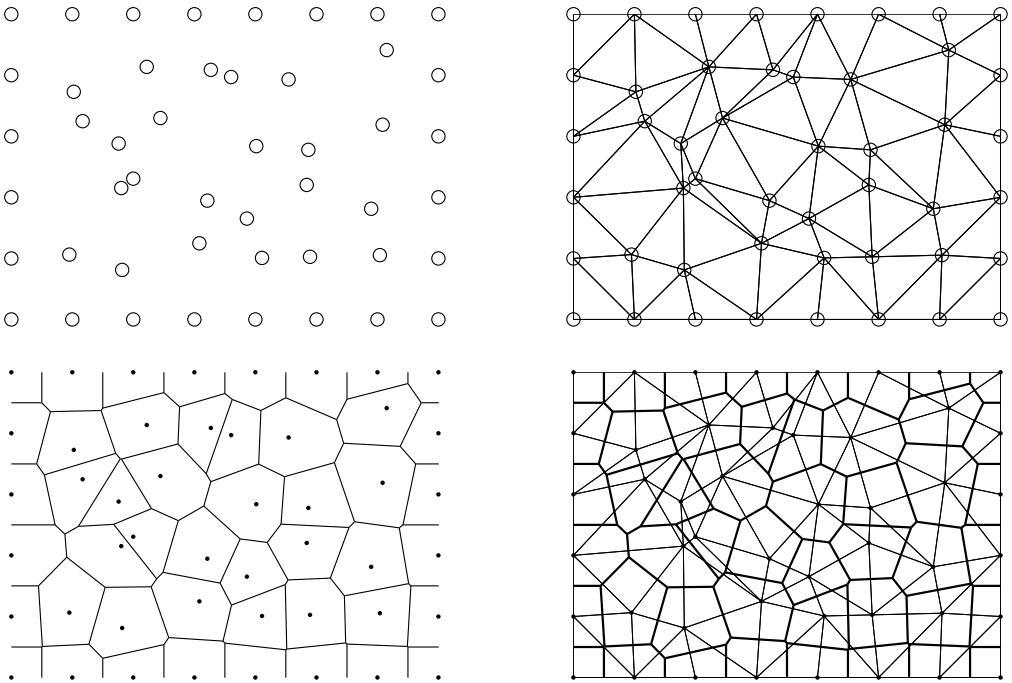
Mapping a curvilinear grid to a complex shape is generally difficult.

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:

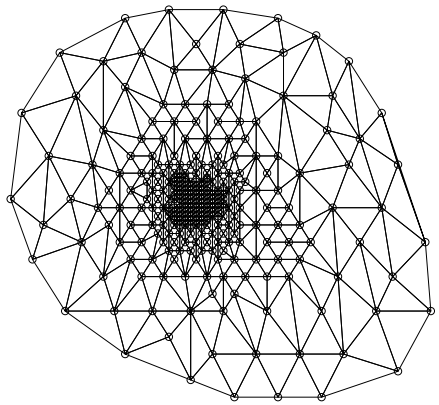
```
x = linspace(-2,2,21);
G = tensorGrid(x,x,x);
subplot(1,2,1); plotGrid(G);view(3); axis equal

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;
```



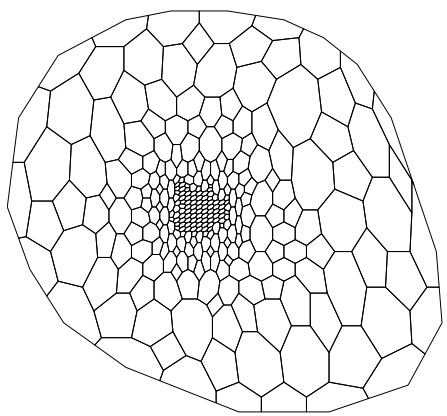
Standard grids: Delaunay and Voronoi grids



Duality: Delaunay and Voronoi grids



```
load seamount; plot(x(:),y(:),'o');
G = triangleGrid([x(:) y(:)]);
plotGrid(G,'FaceColor',[.8 .8 .8]);
```



```
V = pebi(triangleGrid([x y]));
plotGrid(V,'FaceColor',[.8 .8 .8]);
```

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

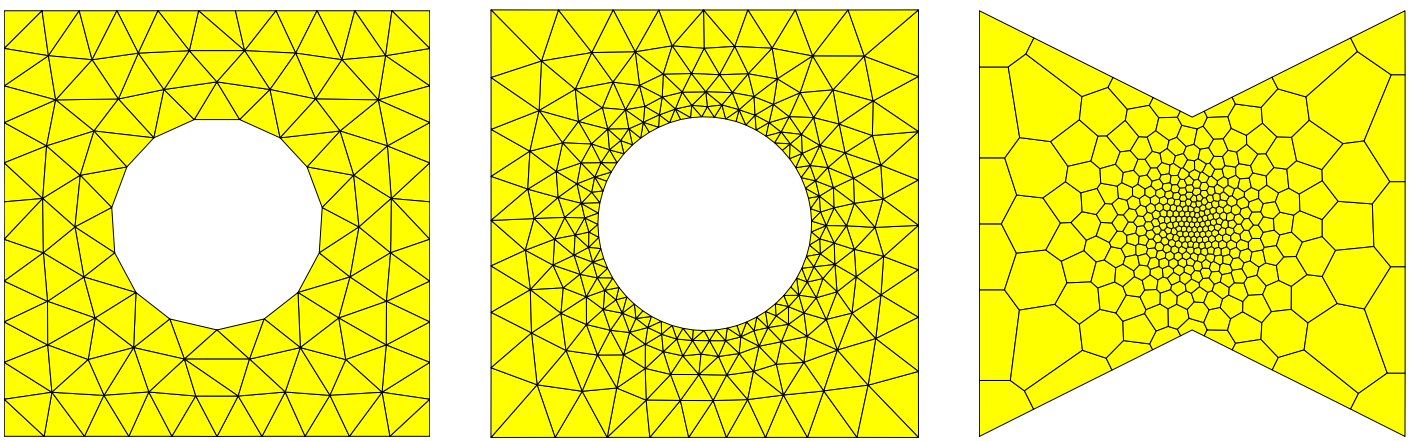
Using an external grid generator

Install `DistMesh` by Persson & Strang as a module

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

and use it to grid around a circular inclusion

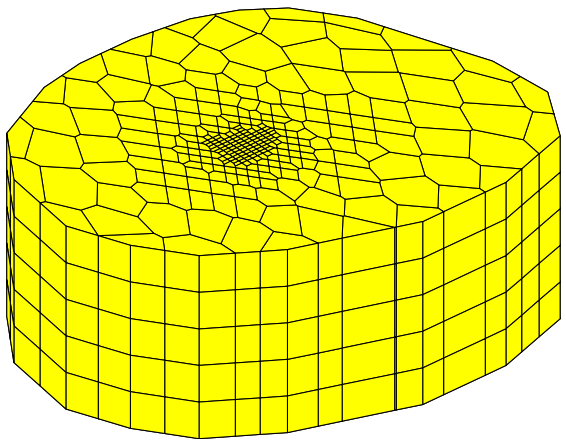
```
mrstModule add distmesh;
fd=@(p) ddiff(drectangle(p,-1,1,-1,1), dcircle(p,0,0,0.5));
[p,t]=distmesh2d(fd, @huniform, 0.2, [-1,-1;1,1], [-1,-1;-1,1;1,-1;1,1]);
G = triangleGrid(p, t);
```



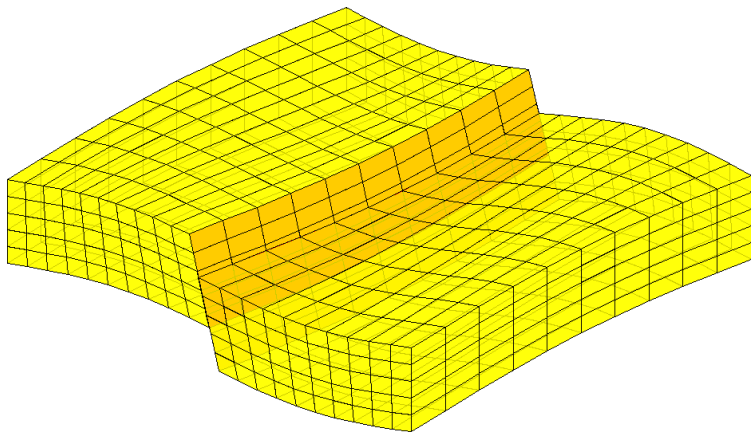
For details: see Chapter 3.2.4 of the MRST book

Layered and stratigraphic grids

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



```
% Extrude a standard MATLAB dataset
load seamount
g = triangleGrid([x(:) y(:)]);
P = pebi(g);
V = makeLayeredGrid(P, 5);
plotGrid(V), view(-40, 60), 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
```

Rock modelling in MRST

All flow and transport solvers in MRST require a rock structure, which by convention is called `rock`, and contains two fields:

rock.poro – porosity, column vector with one entry per active cell

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

$$\mathbf{K}_i = \begin{bmatrix} K_1(i) & K_2(i) \\ K_2(i) & K_3(i) \end{bmatrix}, \qquad \mathbf{K}_i = \begin{bmatrix} K_1(i) & K_2(i) & K_3(i) \\ K_2(i) & K_4(i) & K_5(i) \\ K_3(i) & K_5(i) & K_6(i) \end{bmatrix}$$

Nonsymmetic permeabilities are currently not supported

The `rock` object can also hold net-to-gross, `ntg`, consisting of a scalar or a single column vector with one value per active cell

Example: homogeneous model

Square 10×10 grid model with a uniform porosity of 0.2 and isotropic permeability equal 200 mD:

```
G    = cartGrid([10 10]);
rock = makeRock(G, 200*milli*darcy, 0.2);
```

Because MRST works in SI units, we must convert from the field units 'darcy' to the SI unit 'meters^2'. Alternative: use the conversion function `convertFrom(200,milli*darcy)`

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

```
rock = makeRock(G, [100 100 10].*milli*darcy, .2);
```

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

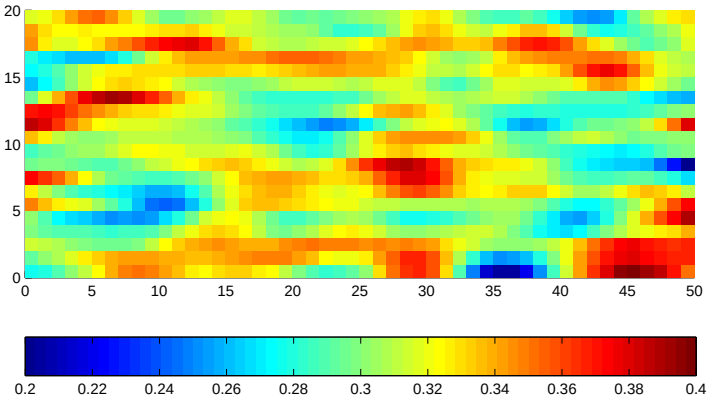
Generate ϕ as a Gaussian field and then compute K from the Carman–Kozeny relation

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

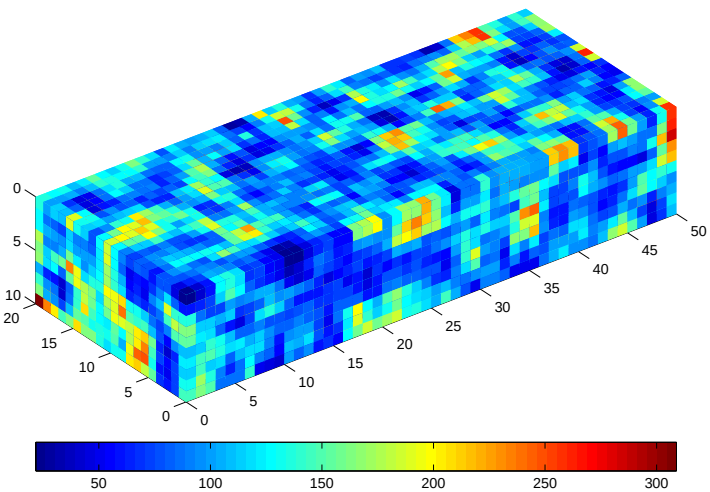
In MRST:

```
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




```
plotCellData(G,rock.poro);
colorbar('horiz'); axis equal tight;
```



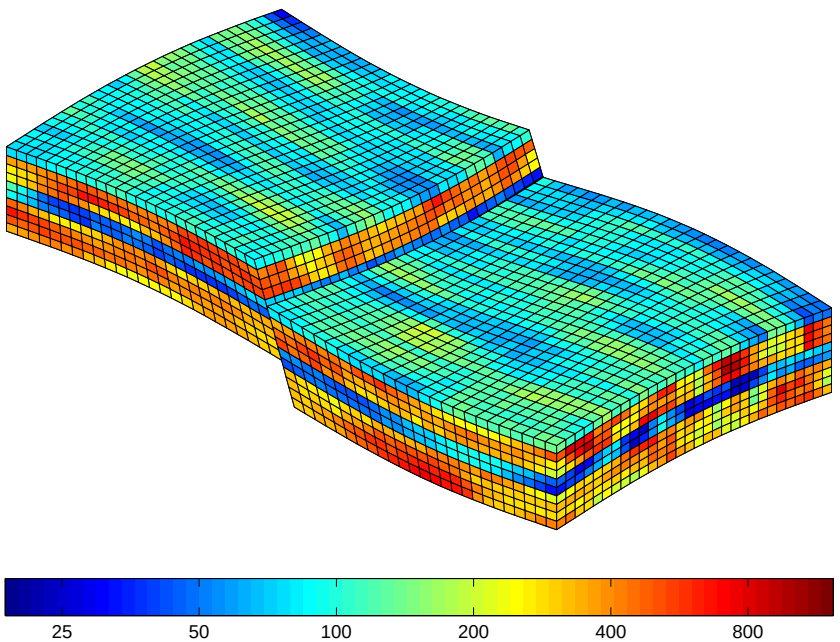

```
plotCellData(G,convertTo(rock.perm,milli*darcy));
colorbar('horiz'); axis equal tight; view(3);
```

Example: stratigraphic model

Generate a layered model with a single fault in the middle.

```
G = processGRDECL(simpleGrdecl([50 30 10], 0.12));
K = logNormLayers(G.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.

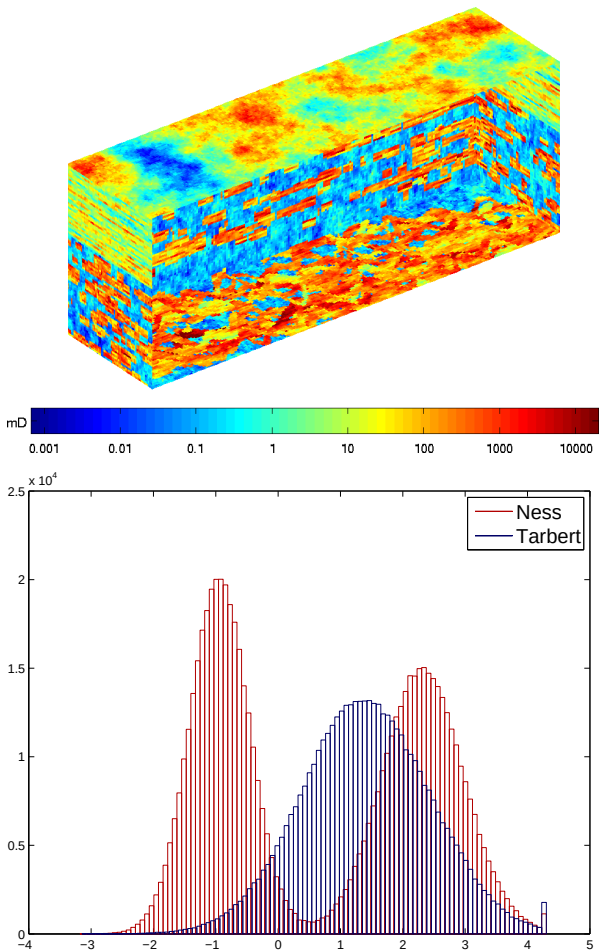



```
plotCellData(G,log10(K),'EdgeColor','k');
view(45,30); axis tight off,
set(gca,'DataAspect',[0.5 1 1])
h     = colorbar('horiz');
ticks = 25*2.^[0:5];
set(h,'XTick',log10(ticks),'XTickLabel',ticks);
```

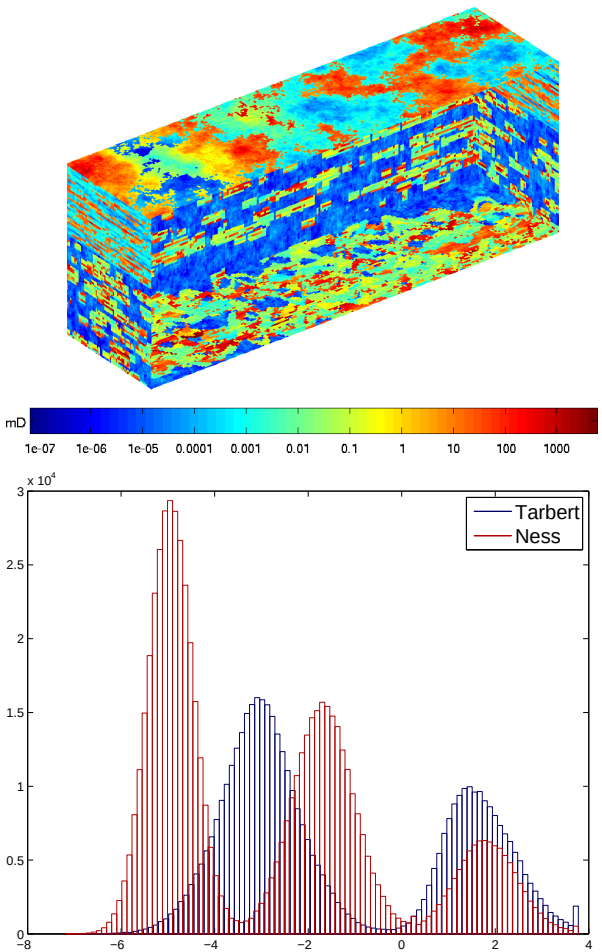
Example: Model 2, 10$^{\text{th}}$ SPE Comparative Solution Project

Separate module, `spe10`, for downloading and accessing this model

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



lateral permeability



vertical permeability

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

```
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]
```

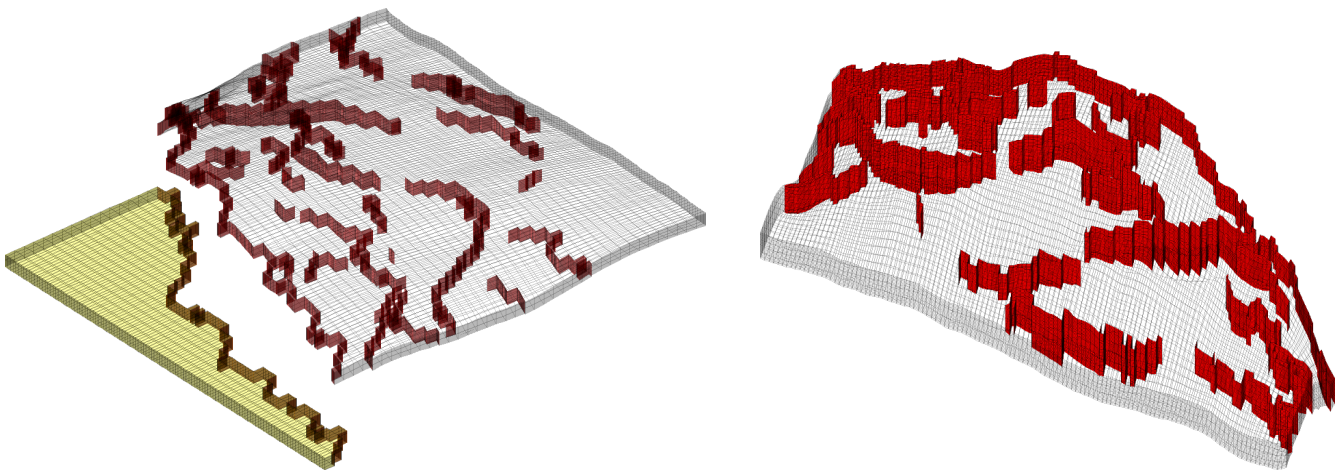
Example: synthetic shallow-marine model

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

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

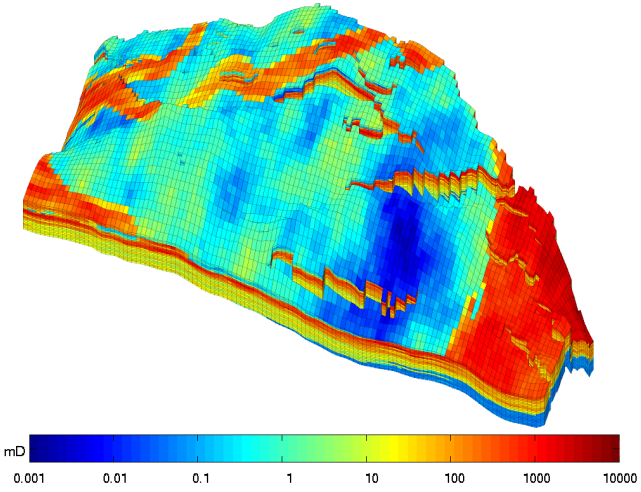
Then we generate a space-filling grid and extract petrophysical properties

```
G = processGRDECL(grdecl);
G = computeGeometry(G);
rock = grdecl2Rock(grdecl, G.cells.indexMap);
```

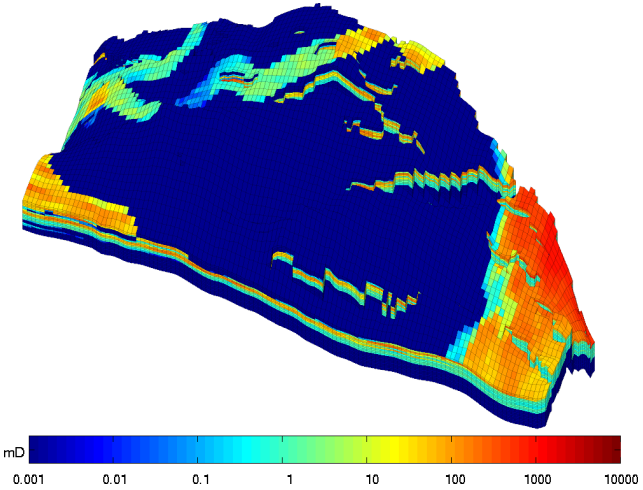


Example: synthetic shallow-marine model

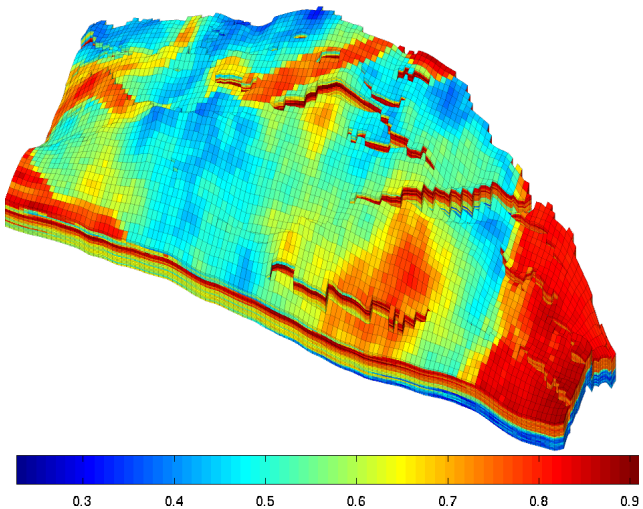
horizontal permeability



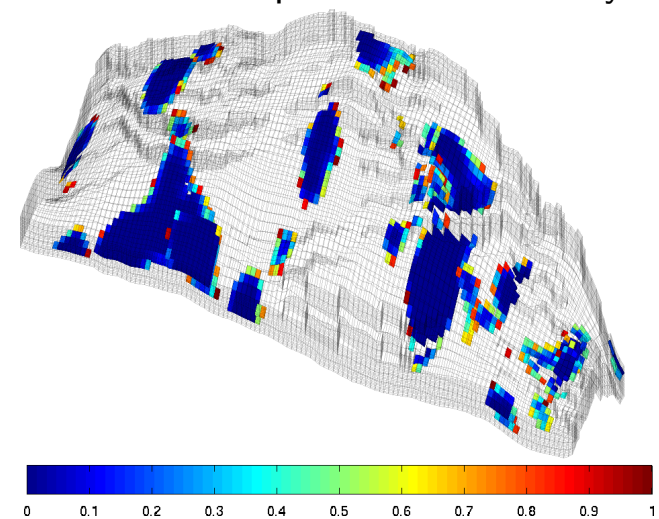
vertical permeability



net-to-gross

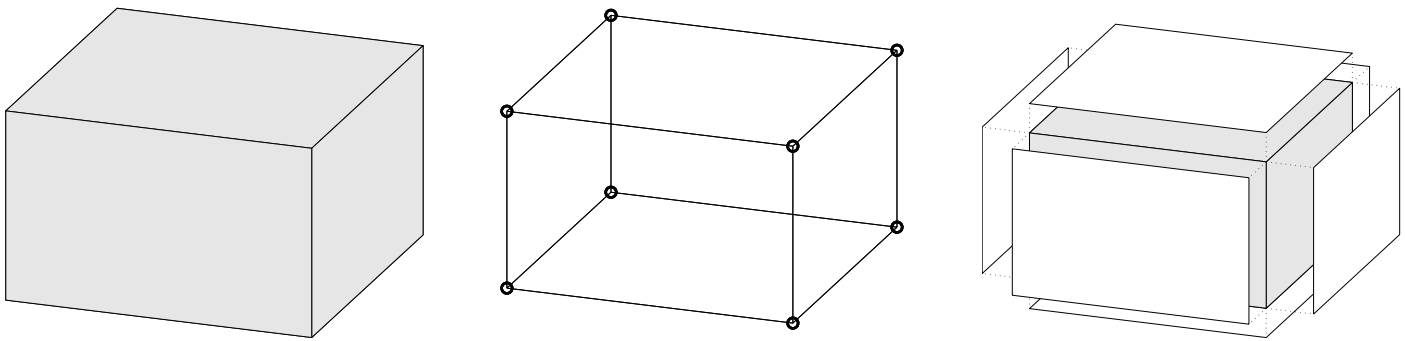


vertical multipliers less than unity



Grids in MRST: fully unstructured

All grids are assumed to be unstructured. Basic representation:



cells nodes faces

- 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

Grid structure: cells

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 `facePos(j+1)-facePos(j)` times. To conserve memory, we regenerate it using run-length encoding:

  ```
  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}] = ind2sub(dims, 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 `nodes(i,1)==j`, local vertex i is part of global face number j and corresponds to global vertex `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 `neighbors(i,1)` and `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 weigthed**, 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.

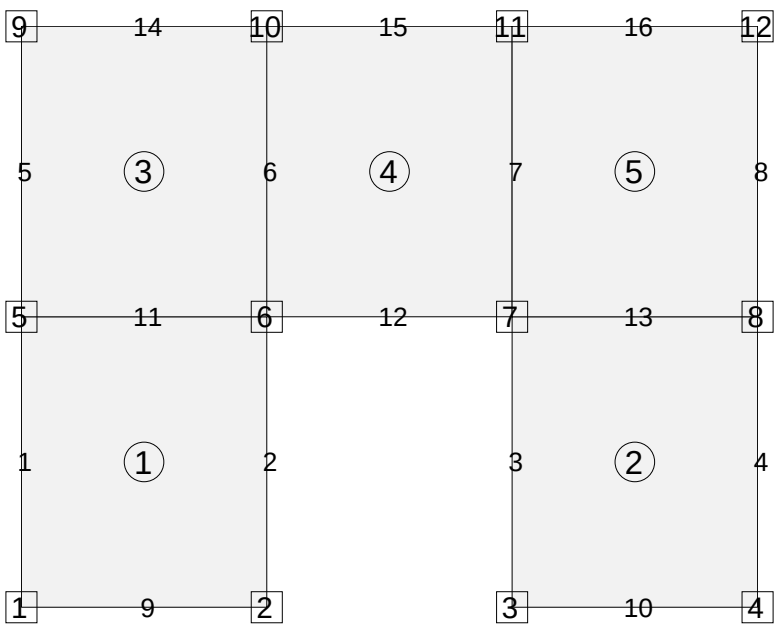
The vertex structure, `G.nodes`, consists of two fields:

- `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 = removeCells( cartGrid([3,2]), 2)
```

```
G =
        cells: [1x1 struct]
        faces: [1x1 struct]
        nodes: [1x1 struct]
    cartDims: [3 2]
         type: {'tensorGrid'  'cartGrid'  'removeCells'}
     griddim: 2
```

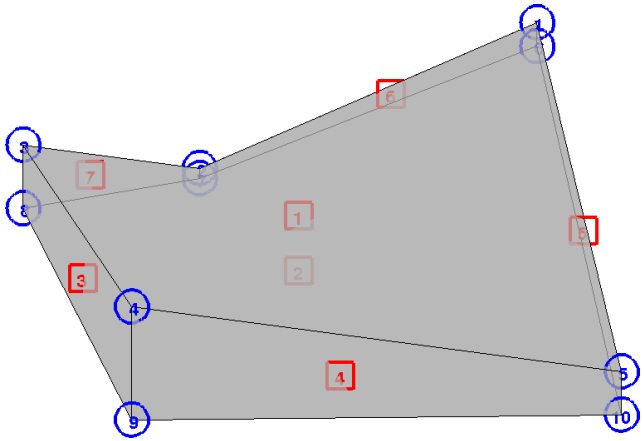



```
cells.faces =
 1   1   1 (East)
 1   9   3 (South)
 1   2   2 (West)
 1  11   4 (North)
 2   3   1 (East)
 2  10   3 (South)
 2   4   2 (West)
 2  13   4 (North)
 3   5   1 (East)
 3  11   3 (South)
 3   6   2 (West)
 3  14   4 (North)
 4   6   1 (East)
 4  12   3 (South)
 4   7   2 (West)
 4  15   4 (North)
 5   7   1 (East)
 5  13   3 (South)
 5   8   2 (West)
 5  16   4 (North)
```

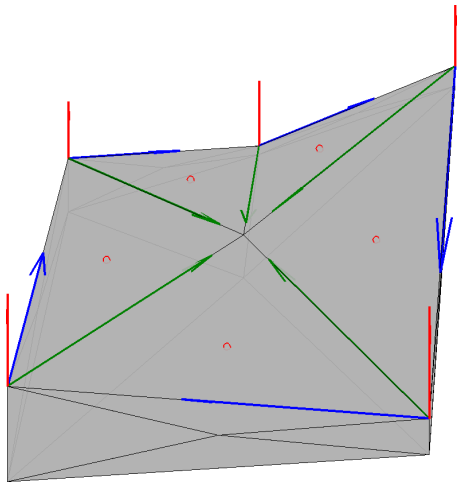
```
faces.nodes =
    1   1
    1   5
    2   2
    2   6
    3   3
    3   7
    4   4
    4   8
    5   5
    5   9
    6   6
    6  10
    7   7
    7  11
    8   8
    8  12
    9   2
    9   1
    :   :
    :   :
```

```
faces.neighbors =
    0   1
    1   0
    0   2
    2   0
    0   3
    3   4
    4   5
    5   0
    0   1
    0   2
    1   3
    0   4
    2   5
    3   0
    4   0
    5   0
```

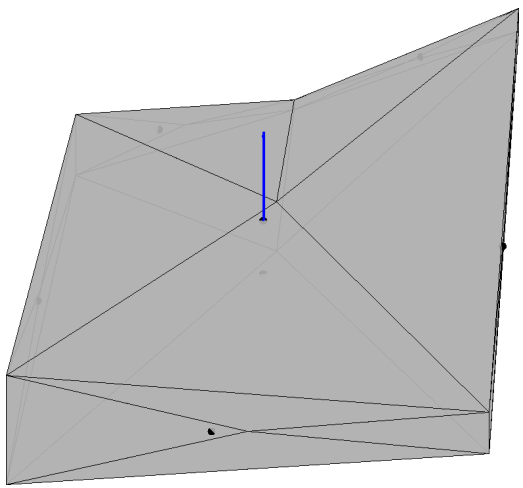
Geometry computation: basic steps



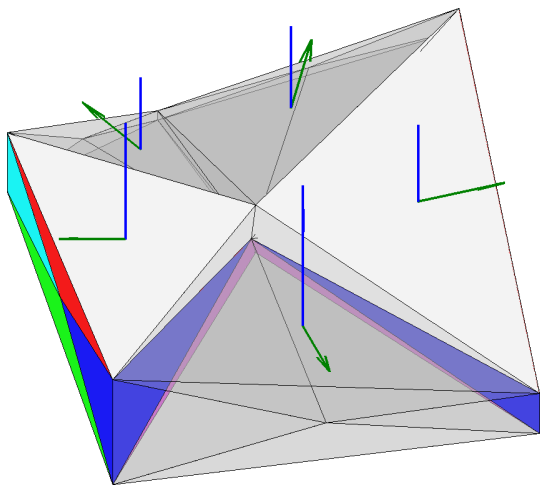
Single cell



Tesselation of faces



Area-weighted centroid and normal vector



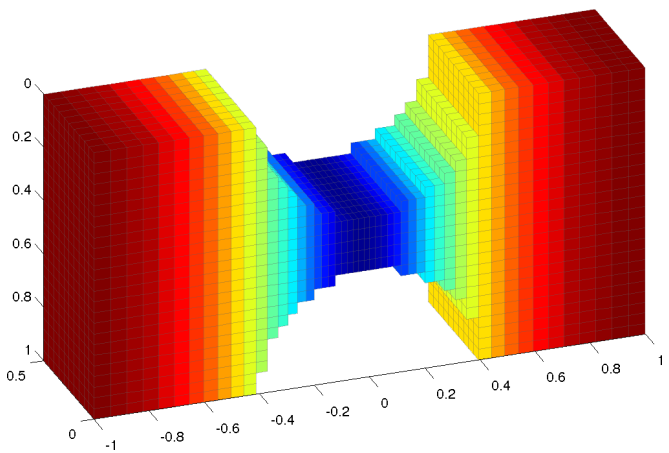
Triangulation of cell volume

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.



- 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

What you will learn in this section

We will go through:

- discrete differential and averaging operators
- finite volume methods for $-\nabla\left(\mathbf{K}\nabla p\right) = q$
- automatic differentiation
- flow solvers in the `incomp` family

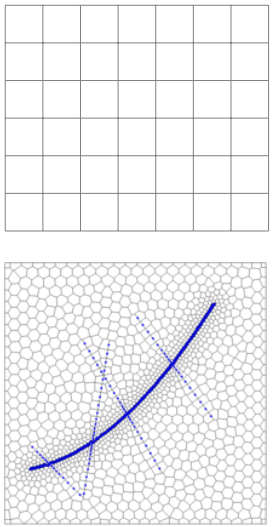
You will also get a tast of efficient vectorization tricks in MATLAB

To learn more:

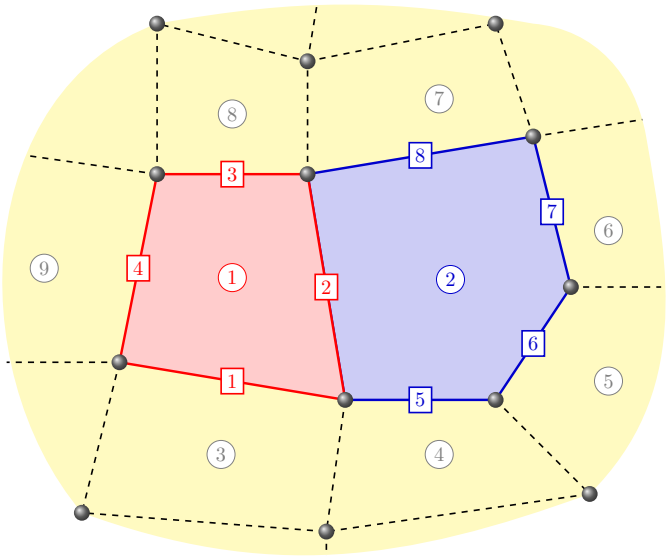
- watch the videos in Jolt1
- study the `1ph` tutorials/examples in the `incomp` module
- read Lie et al. (COMG, 2012), doi: 10.1007/s10596-011-9244-4
- read Chapters 4 to 6 in the MRST book

Discrete differentiation operators

Idealized models



Grid structure in MRST



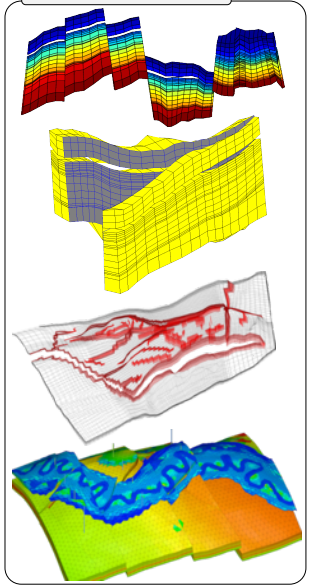
c	F(c)
1	1
1	2
1	3
1	4
2	5
2	6
2	7
2	8
2	2
3	1
⋮	⋮
⋮	⋮

Map: cell → faces

f	C_1	C_2
1	3	1
2	1	2
3	1	8
4	9	1
5	4	2
6	2	5
7	2	6
8	2	7
⋮	⋮	⋮
⋮	⋮	⋮

Map: face → cells

Industry models



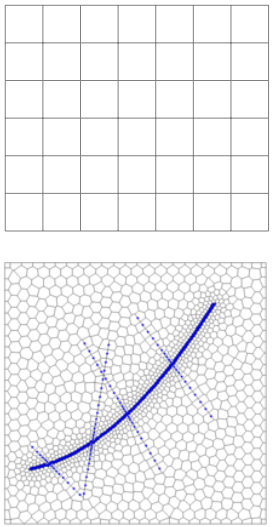
Starting point: mapping F from cell to faces, and C from face to cells:

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

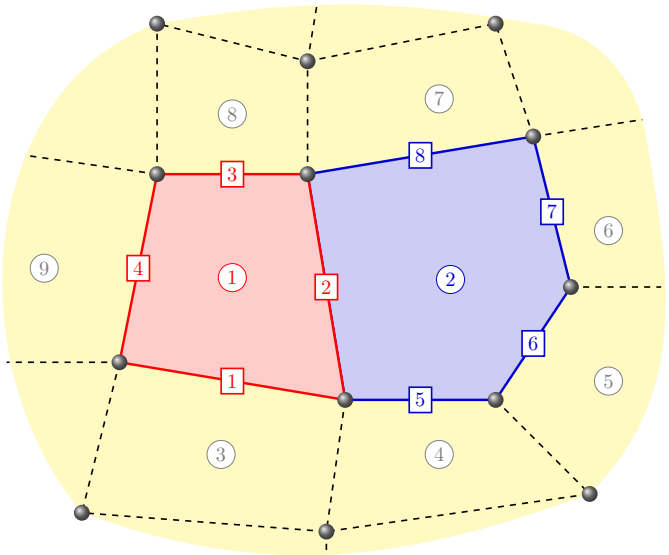
We only consider internal faces, mapping F is represented as `cn` and `F`

Discrete differentiation operators

Idealized models



Grid structure in MRST



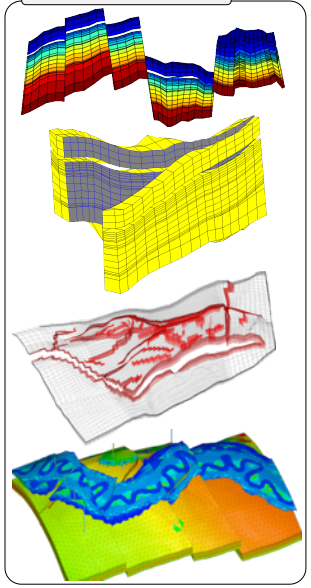
c	F(c)
1	1
1	2
1	3
1	4
2	5
2	6
2	7
2	8
2	2
3	1
⋮	⋮
⋮	⋮

Map: cell → faces

f	C_1	C_2
1	3	1
2	1	2
3	1	8
4	9	1
5	4	2
6	2	5
7	2	6
8	2	7
⋮	⋮	⋮
⋮	⋮	⋮

Map: face → cells

Industry models



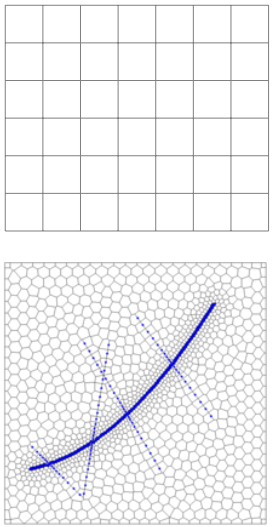
The discrete `div` operator is a linear mapping from faces to cells:

$$\texttt{div}(\boldsymbol{v})[c] = \sum_{f \in F(c)} \operatorname{sgn}(f)\boldsymbol{v}[f], \qquad \operatorname{sgn}(f) = \begin{cases} 1, & \text{if } c = C_1(f), \\ -1, & \text{if } c = C_2(f). \end{cases}$$

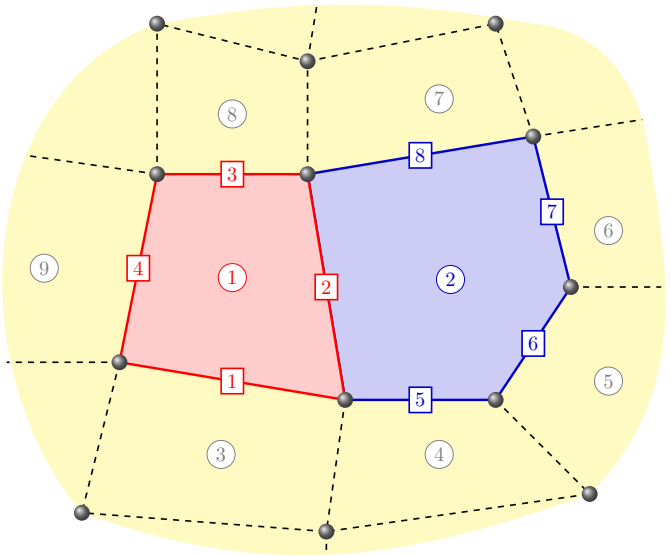
Here, $\boldsymbol{v}[f]$ denotes a discrete flux over face f with orientation from cell $C_1(f)$ to cell $C_2(f)$

Discrete differentiation operators

Idealized models



Grid structure in MRST



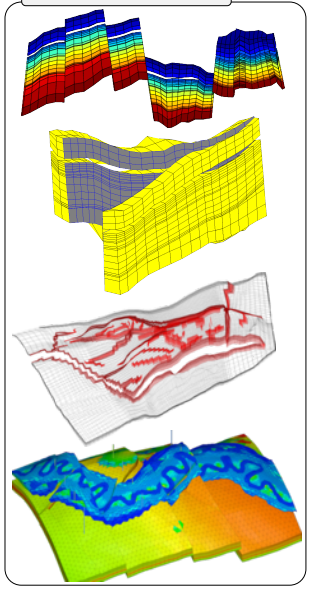
c	F(c)
1	1
1	2
1	3
1	4
2	5
2	6
2	7
2	8
2	2
3	1
⋮	⋮
⋮	⋮

Map: cell → faces

f	C_1	C_2
1	3	1
2	1	2
3	1	8
4	9	1
5	4	2
6	2	5
7	2	6
8	2	7
⋮	⋮	⋮
⋮	⋮	⋮

Map: face → cells

Industry models



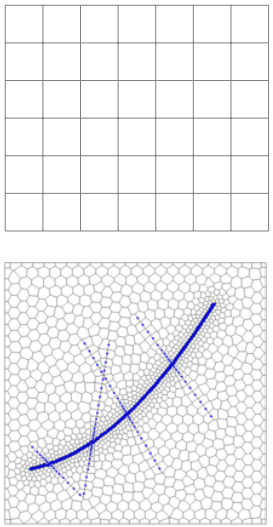
The discrete `grad` operator maps from cell pair $C_1(f), C_2(f)$ to face f:

$$\texttt{grad}(\boldsymbol{p})[f] = \boldsymbol{p}[C_2(f)] - \boldsymbol{p}[C_1(f)],$$

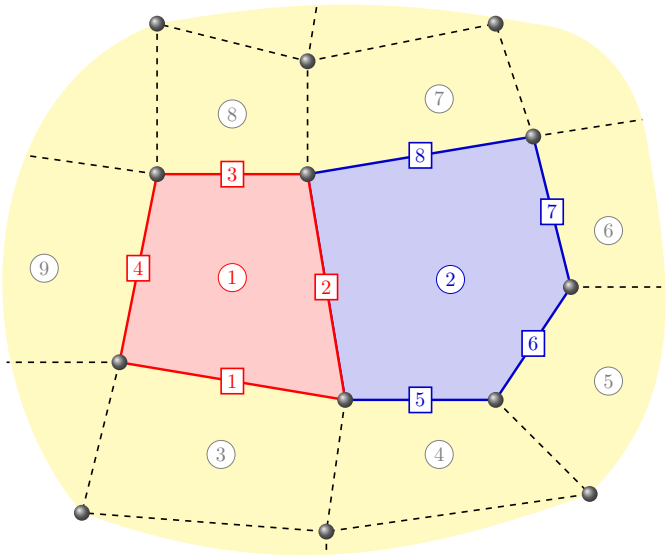
where $\boldsymbol{p}[c]$ is a scalar quantity associated with cell c

Discrete differentiation operators

Idealized models



Grid structure in MRST



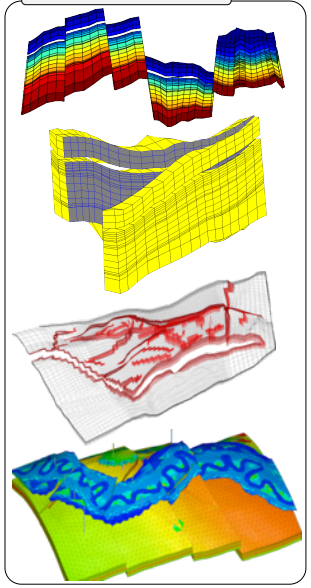
c	F(c)
1	1
1	2
1	3
1	4
2	5
2	6
2	7
2	8
2	2
3	1
⋮	⋮
⋮	⋮

Map: cell → faces

f	C_1	C_2
1	3	1
2	1	2
3	1	8
4	9	1
5	4	2
6	2	5
7	2	6
8	2	7
⋮	⋮	⋮
⋮	⋮	⋮

Map: face → cells

Industry models



The `div` and `grad` operators are linear and can be represented as sparse matrix multiplications:

```
D = sparse([(1:nf)'; (1:nf)'], C, ones(nf,1)*[-1 1], nf, nc);
grad = @(x) D*x;
div  = @(x) -D'*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

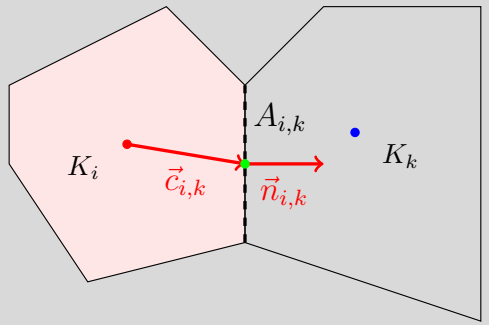
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

$$\boldsymbol{v}[f] = -\boldsymbol{T}[f] \, \mathtt{grad}(\boldsymbol{p})[f]$$

Conservation of mass:

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

$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



$$T_{i,k} = A_{i,k} \frac{\vec{c}_{i,k} \cdot \mathbf{K}_i \vec{n}_{i,k}}{|\vec{c}_{i,k}|^2}$$

$$T_{ik} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}$$

Finite-volume method: single-phase flow

Fundamental physics: Darcy's law

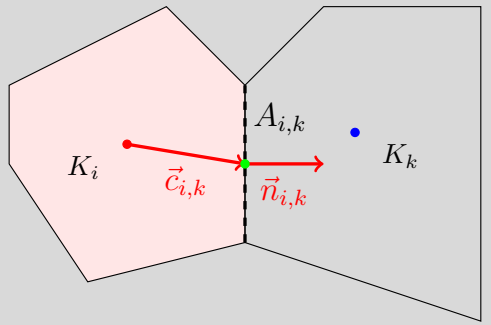
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

$$\boldsymbol{v}[f] = -\boldsymbol{T}[f] \, \mathtt{grad}(\boldsymbol{p})[f]$$

Conservation of mass:

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

$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



$$T_{i,k} = A_{i,k} \frac{\vec{c}_{i,k} \cdot \mathbf{K}_i \vec{n}_{i,k}}{|\vec{c}_{i,k}|^2}$$

$$T_{ik} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}$$

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

```
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

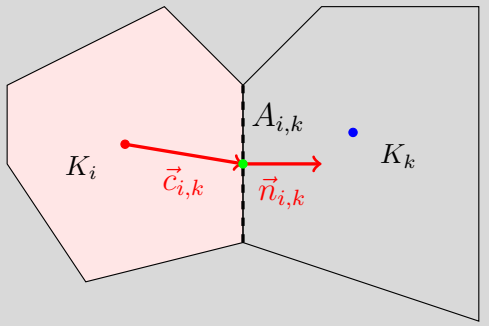
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

$$\boldsymbol{v}[f] = -\boldsymbol{T}[f] \, \mathtt{grad}(\boldsymbol{p})[f]$$

Conservation of mass:

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

$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



$$T_{i,k} = A_{i,k} \frac{\vec{c}_{i,k} \cdot \mathbf{K}_i \vec{n}_{i,k}}{|\vec{c}_{i,k}|^2}$$

$$T_{ik} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}$$

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

```
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

Extract permeability vector $[K_{xx}, K_{xy}, K_{yx}, K_{yy}]$ for each cell:

```
[K, i, j] = permTensor(rock, G.griddim);
```

Finite-volume method: single-phase flow

Fundamental physics: Darcy's law

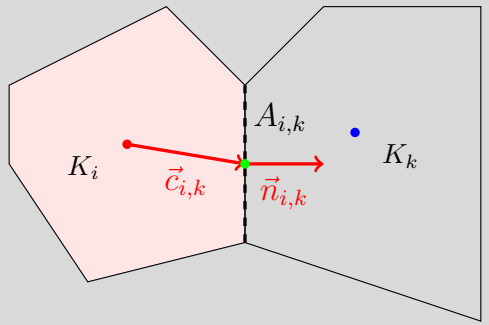
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

$$\boldsymbol{v}[f] = -\boldsymbol{T}[f] \, \mathtt{grad}(\boldsymbol{p})[f]$$

Conservation of mass:

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

$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



$$T_{i,k} = A_{i,k} \frac{\vec{c}_{i,k} \cdot \mathbf{K}_i \vec{n}_{i,k}}{|\vec{c}_{i,k}|^2}$$

$$T_{ik} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}$$

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

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

Here, `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

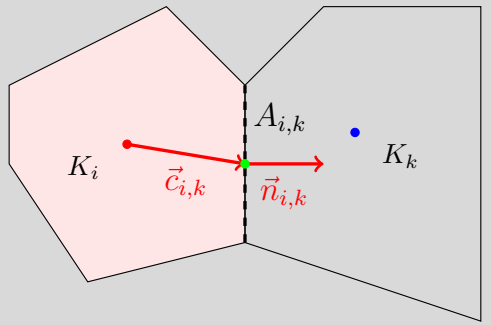
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

$$\boldsymbol{v}[f] = -\boldsymbol{T}[f] \, \mathtt{grad}(\boldsymbol{p})[f]$$

Conservation of mass:

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

$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



$$T_{i,k} = A_{i,k} \frac{\vec{c}_{i,k} \cdot \mathbf{K}_i \vec{n}_{i,k}}{|\vec{c}_{i,k}|^2}$$

$$T_{ik} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}$$

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

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

Here, `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 = computeTrans(G, rock);
```

Finite-volume method: single-phase flow

Fundamental physics: Darcy's law

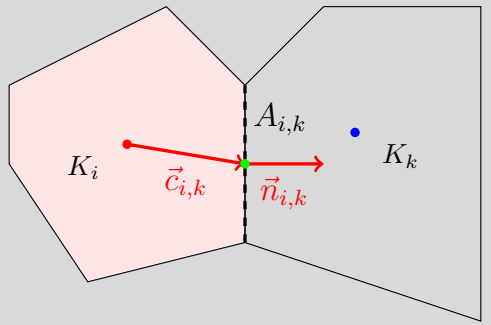
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

$$\boldsymbol{v}[f] = -\boldsymbol{T}[f] \, \mathtt{grad}(\boldsymbol{p})[f]$$

Conservation of mass:

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

$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



$$T_{i,k} = A_{i,k} \frac{\vec{c}_{i,k} \cdot \mathbf{K}_i \vec{n}_{i,k}}{|\vec{c}_{i,k}|^2}$$

$$T_{ik} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}$$

Finally, we can compute the transmissibilities T_{ik}:

```
T  = 1 ./ accumarray(F, 1 ./ hT, [G.faces.num, 1]);
T  = T(all(C~=0,2),:);
```

Here, `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

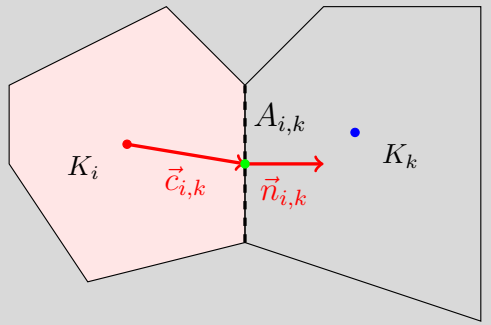
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

$$\boldsymbol{v}[f] = -\boldsymbol{T}[f] \, \mathtt{grad}(\boldsymbol{p})[f]$$

Conservation of mass:

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

$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



$$T_{i,k} = A_{i,k} \frac{\vec{c}_{i,k} \cdot \mathbf{K}_i \vec{n}_{i,k}}{|\vec{c}_{i,k}|^2}$$

$$T_{ik} = [T_{i,k}^{-1} + T_{k,i}^{-1}]^{-1}$$

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

```
S = setupOperatorsTPFA(G,rock);
```

```
S =
      T_all: [220x1 double]
          T: [180x1 double]
          C: [180x100 double]
       Grad: @(x)-C*x
        Div: @(x)C'*x
            :
```

Automatic differentiation

Discretization of flow models leads to large system of (non)linear equations. Can be linearized and solved with Newton's method

$$\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0} \qquad \Rightarrow \qquad \frac{\partial \boldsymbol{F}}{\partial \boldsymbol{x}}(\boldsymbol{x}^i)(\boldsymbol{x}^{i+1} - \boldsymbol{x}^i) = -\boldsymbol{F}(\boldsymbol{x}^i)$$

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 ($+$, $-$, $*$, $/$,...) and elementary functions (`sin`, `exp`, `power`, ...)
- Derivative rules are known for these operations and functions
- Combine these with the chain rule

Automatic differentiation

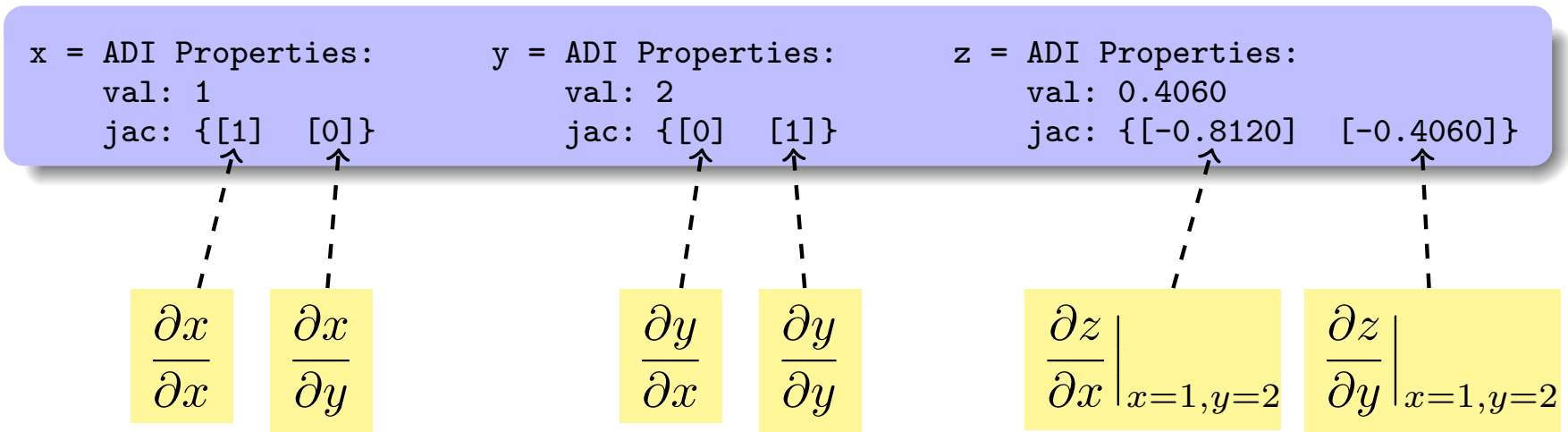
- Consider a scalar primary variable x and a function $f(x)$, whose AD representations are the pairs $\langle x, 1 \rangle$ and $\langle f, f_x \rangle$
- Must define the action of elementary operations and functions on all such pairs:

$$\begin{aligned}
\langle f, f_x \rangle + \langle g, g_x \rangle &= \langle f + g, f_x + g_x \rangle \,, \\
\langle f, f_x \rangle \langle g, g_x \rangle &= \langle fg, fg_x + f_x g \rangle \,, \\
\exp \langle \langle f, f_x \rangle \rangle &= \langle \exp \langle f \rangle \,, \exp \langle f \rangle f_x \rangle \,, \\
\sin \langle \langle f, f_x \rangle \rangle &= \langle \sin \langle f \rangle \,, \cos \langle f \rangle f_x \rangle \,,
\end{aligned}$$

- Use operator overloading to write $a + b * c$ 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] = initVariablesADI(1,2);
z = 3*exp(-x*y)
```



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:

$$F(p) = \nabla \cdot \big(\mathbf{K} \nabla p\big) + q = 0$$

In discrete form:

$$\boldsymbol{F}(\boldsymbol{p}) = \texttt{div}\big(\boldsymbol{T}\,\texttt{grad}(\boldsymbol{p})\big) + \boldsymbol{q} = \boldsymbol{A}\boldsymbol{p} + \boldsymbol{q} = \boldsymbol{0}$$

Apply Newton's method with a zero initial guess:

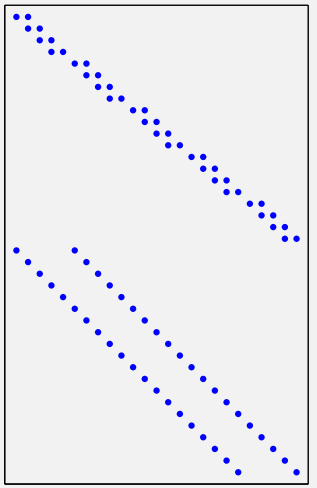
$$\frac{\partial \boldsymbol{F}}{\partial \boldsymbol{p}}(\boldsymbol{p})\big(\boldsymbol{p} - \boldsymbol{0}\big) = -\boldsymbol{F}(\boldsymbol{0}) \qquad \Leftrightarrow \qquad \boldsymbol{A}\boldsymbol{p} = -\boldsymbol{q}$$

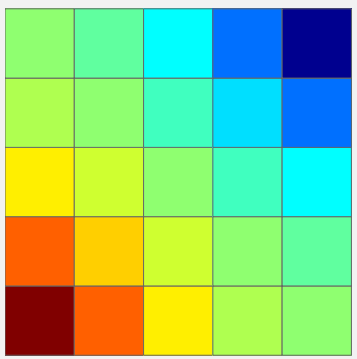
Using AD means that we never need to form $\boldsymbol{A}$ explicitly

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);
```

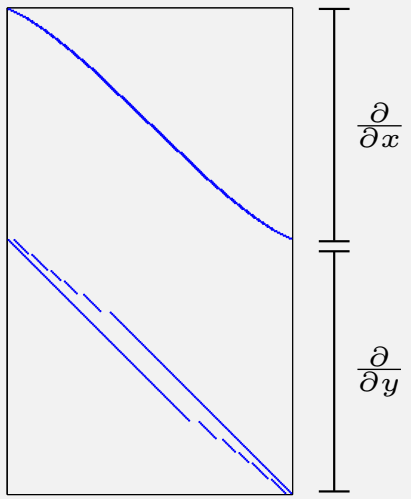
$C =$

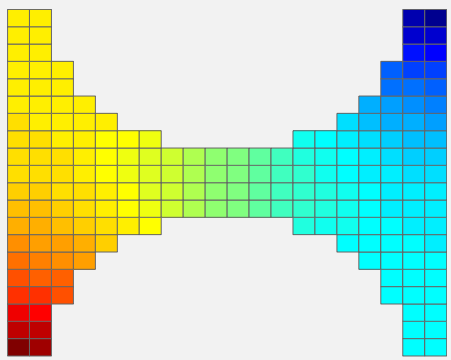




Solving the Poisson equation: non-rectangular domain

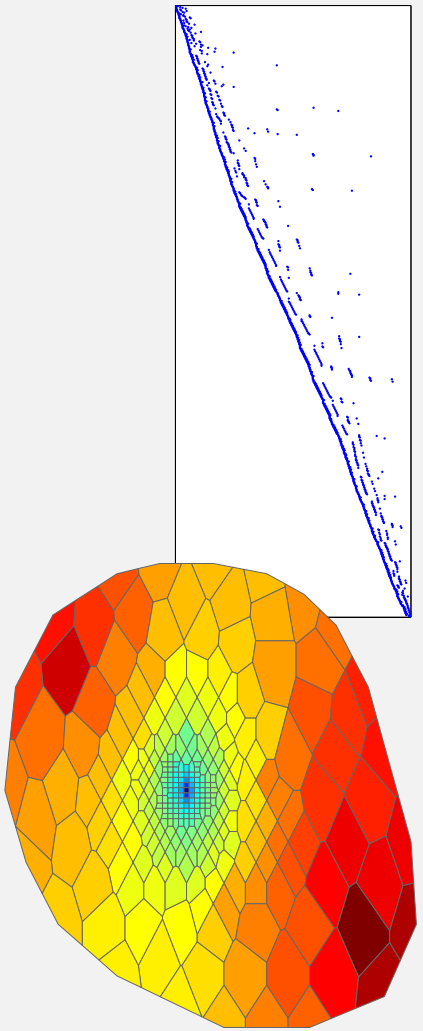
```
% Grid and grid information
G    = computeGeometry(cartGrid([20 20], [1 1]));
r1   = sum(bsxfun(@minus, G.cells.centroids ,[0 .5  1]) . ^2,2);
r2   = sum(bsxfun(@minus, G.cells.centroids ,[0 .5  0]) . ^2,2);
G    = extractSubgrid(G, (r1>0.16) & (r2>0.16));
rock = makeRock(G, 1, 1);
nc   = G.cells.num;

% 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 .5 .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 dicretization schemes

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

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

Two-point flux approximation (TPFA)

Given a *vector* $\mathbf{T}_{tp}$ of transmissibilities, the coded equations become

```
v  = -T_tp.*grad(p)
eq = div(v)-q;
```

$\longleftrightarrow$

```
eq = div(T_tp.*grad(p))+q;
```

Multi-point flux approximation (MPFA)

Given a *matrix* $\mathbf{T}_{mp}$ of transmissibilities, the coded equations become

```
v  = -T_mp*grad(p)
eq = div(v)-q;
```

$\longleftrightarrow$

```
eq = div(T_mp*grad(p))+q;
```

Switching between different dicretization schemes, cont'd

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

Lowest order mixed (or mimetic) formulation

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

```
v  = -M\grad(p);
eq = div(v)-q;
```

... 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:

```
[v,p] = initVariablesADI(zeros(nf, 1), zeros(nc, 1));
eq{1} = M*v+grad(p);          % Darcy's law
eq{2} = div(v)-q;             % continuity equation
eq    = cat(eq{:});           % concatenate equations
x     = -eq.jac{1}\eq.val;    % solve, x contains both v and p
```

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 `incomp` family of solvers:

`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 diagnonal:

```
d = accumarray([C(:,1); C(:,2)], 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]; clear d;
A = 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);          % Set p=0 in cell #1 if only Dirichlet b.c
p = mldivide(A, 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

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

For grids having logical IJK numbering:

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

where `side` would be `1/'West'/'XMin'/'Left'`, etc

- Wells with Peacemann well model:

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

For convenience, we also have the functions

```
W = verticalWell(W, G, rock, I, J, K)
W = verticalWell(W, G, rock, I,    K)
```

Incompressible flow solvers in MRST

The three main solvers available are:

- The standard two-point solver:

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

- Lowest-order mimetic finite-difference methods:

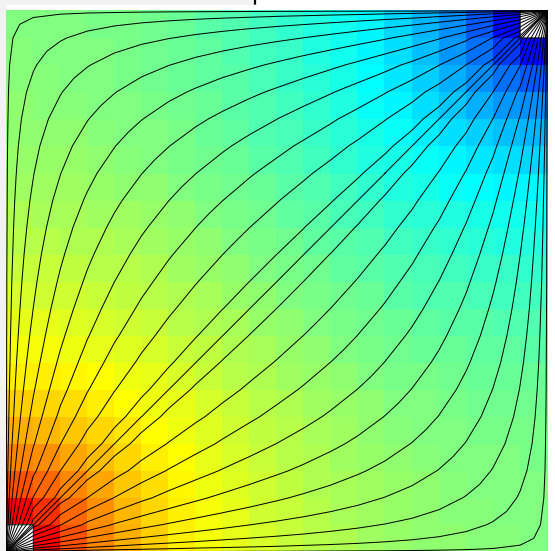
```
mrstModule add mimetic;
IP = computeMimeticIP(G, rock);
state = incompMimetic(state, G, IP, fluid, ... );
```

- The MPFA-O multipoint flux-approximation method:

```
mrstModule add mpfa;
hT = computeMultiPointTrans(G, rock);
state = incompMPFA(state, G, hT, fluid)
```

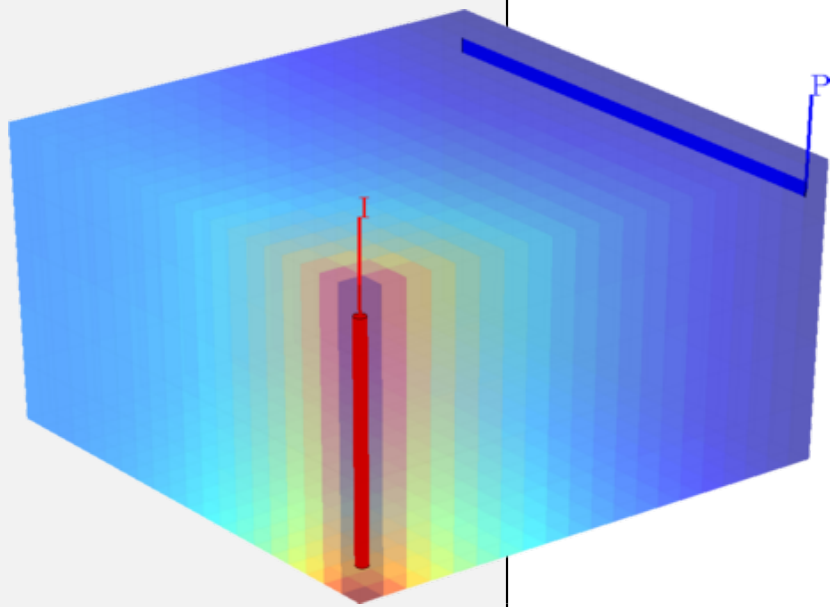
Example: quarter five-spot with source terms

```
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] = deal(20,20,5);
G  = cartGrid([nx,ny,nz], [500 500 25])
G  = computeGeometry( );
   :
hT = computeTrans(G, rock);

W = verticalWell([], G, rock, 1, 1, 1:nz, ...
        'Type', 'rate','Comp_i', 1, ...
        'Val', 3e3/day, ... .
        'Radius', .12*meter, 'name', 'I');
W = addWell(W, G, rock, nx : ny : nx*ny, ...
        'Type', 'bhp', 'Comp_i', 1, ...
         'Val', 1.0e5, 'Radius', .12*meter,...
        'Dir', 'y', 'name', 'P');

gravity reset on;
state = initState(G, W, 0);
state = incompTPFA(state, G, hT, fluid, 'wells', W);

plotCellData(G, state.pressure,'EdgeAlpha',.01,'FaceAlpha',.4);
plotWell(G, W(1), 'radius', 1, 'color', 'r');
plotWell(G, W(2), 'radius', .5, 'color', 'b');
view(3), camproj perspective, axis tight off
```



Computer exercises

- 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`

Outline

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

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

and a hyperbolic/parabolic transport equation

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \big[f_w \big(\vec{v} + \boldsymbol{\lambda}_n (\Delta \rho g \nabla z + \nabla P_c) \big) \big] = q_w$$

Two-phase, incompressible flow

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

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

and a hyperbolic/parabolic transport equation

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \big[f_w \big(\vec{v} + \boldsymbol{\lambda}_n (\Delta \rho g \nabla z + \nabla P_c) \big) \big] = q_w$$

Standard approach – sequential solution procedure:

```
Compute initial state and set t = 0
While t < T
    Fix S_w and solve elliptic pressure equation
    Fix p and v and solve transport equation a time Δt
    t = t + Δt
```

Two-phase, incompressible flow

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

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

and a hyperbolic/parabolic transport equation

$$\phi \frac{\partial S_w}{\partial t} + \nabla \cdot \big[f_w \big(\vec{v} + \boldsymbol{\lambda}_n (\Delta \rho g \nabla z + \nabla P_c) \big) \big] = q_w$$

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

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

and a hyperbolic/parabolic transport equation

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

For the transport equation, we have two different solvers:

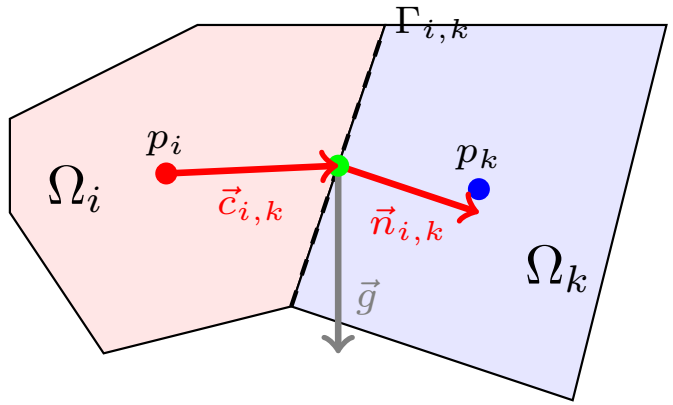
```
state = explicitTransport(state, G, tf, rock, fluid, 'mech1', obj1, ...)
state = implicitTransport(state, G, tf, rock, fluid, 'mech1', obj1, ...)
```

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:

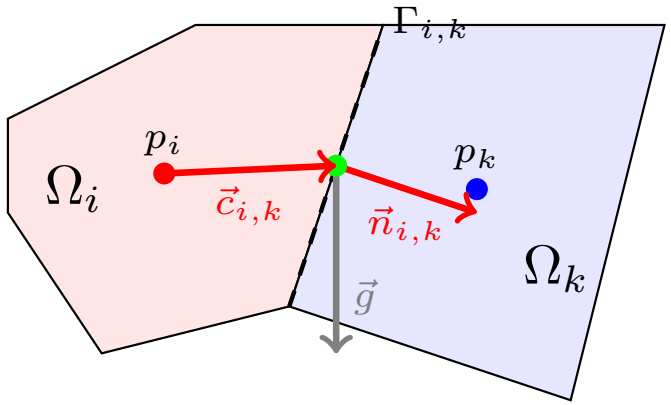
$$\begin{aligned}\vec{H}(S) &= f(s)\vec{v} + \frac{\lambda_w \lambda_n \mathbf{K}}{\lambda_w + \lambda_n}\left(\Delta\rho\,\vec{g} + \nabla P_c(S)\right) \\ &= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).\end{aligned}$$



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

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

$$\begin{aligned}\vec{H}(S) &= f(s)\vec{v} + \frac{\lambda_w \lambda_n \mathbf{K}}{\lambda_w + \lambda_n}\left(\Delta\rho\,\vec{g} + \nabla P_c(S)\right) \\ &= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).\end{aligned}$$



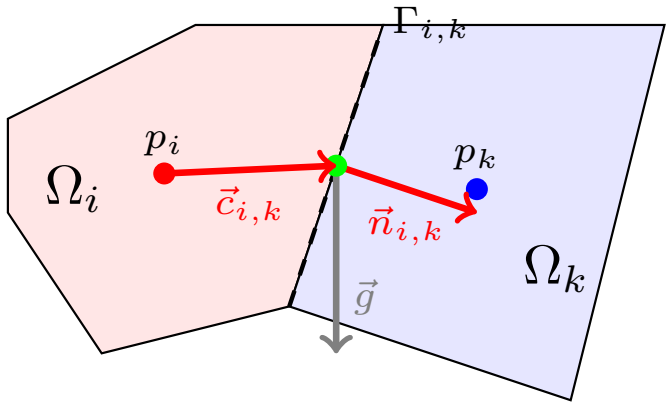
Integrated over cell Ω_i and in time

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

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

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

$$\begin{aligned}\vec{H}(S) &= f(s)\vec{v} + \frac{\lambda_w \lambda_n \mathbf{K}}{\lambda_w + \lambda_n}\left(\Delta\rho\,\vec{g} + \nabla P_c(S)\right) \\ &= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).\end{aligned}$$



Integrated over cell Ω_i and in time

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

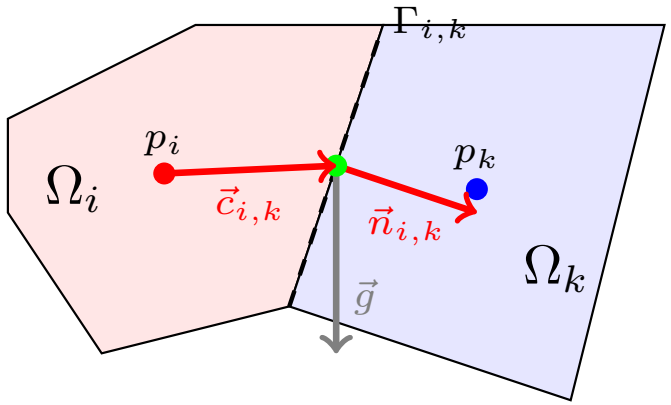
For first-order methods, we can evaluate integral at end-points:

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

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

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

$$\begin{aligned}\vec{H}(S) &= f(s)\vec{v} + \frac{\lambda_w \lambda_n \mathbf{K}}{\lambda_w + \lambda_n}\left(\Delta\rho\,\vec{g} + \nabla P_c(S)\right) \\ &= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).\end{aligned}$$



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

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

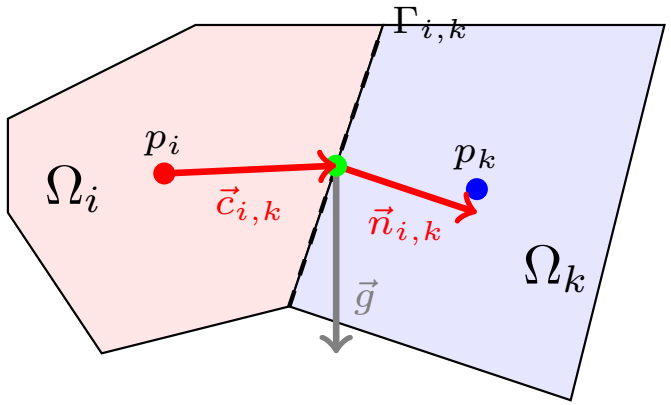
Capillary term is discretized using the TPFA method:

$$A_{i,k} \nabla P_c(S) \cdot \vec{n}_{i,k} \approx \left[T_{i,k}^{-1} + T_{k,i}^{-1}\right]^{-1} \big(P_c(S_i) - P_c(S_k)\big) = P_{i,k}(S)$$

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

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

$$\begin{aligned}\vec{H}(S) &= f(s)\vec{v} + \frac{\lambda_w \lambda_n \mathbf{K}}{\lambda_w + \lambda_n}\left(\Delta\rho\,\vec{g} + \nabla P_c(S)\right) \\ &= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).\end{aligned}$$



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

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

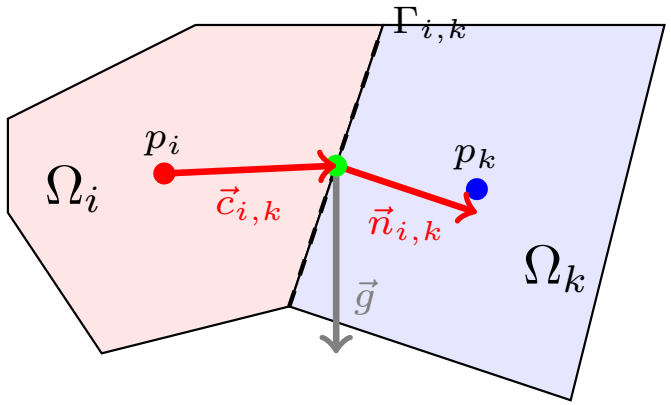
We define a "gravity flux" g_{ik} that is independent of saturation:

$$g_{ik} = \left[g_{i,k}^{-1} + g_{k,i}^{-1}\right]^{-1}, \qquad \begin{aligned} g_{i,k} &= (\Delta\rho)|_{\Omega_i}\, \mathbf{K}_i \vec{g} \cdot \vec{n}_{i,k}, \\ g_{k,i} &= (\Delta\rho)|_{\Omega_k}\, \mathbf{K}_k \vec{g} \cdot \vec{n}_{k,i} \end{aligned}$$

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

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

$$\begin{aligned}\vec{H}(S) &= f(s)\vec{v} + \frac{\lambda_w \lambda_n \mathbf{K}}{\lambda_w + \lambda_n}\left(\Delta\rho\,\vec{g} + \nabla P_c(S)\right) \\ &= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).\end{aligned}$$



Summing up, we have:

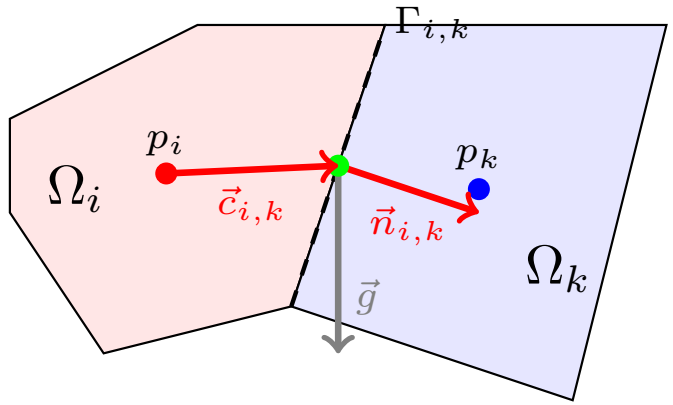
$$H_{ik} = \frac{\lambda_w^u}{\lambda_w^u + \lambda_n^u} v_{ik} + \frac{\lambda_w^u \lambda_n^u}{\lambda_w^u + \lambda_n^u}\left[g_{ik} + P_{ik}\right]$$

where λ_w^u and λ_n^u 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:

$$\begin{aligned}\vec{H}(S) &= f(s)\vec{v} + \frac{\lambda_w \lambda_n \mathbf{K}}{\lambda_w + \lambda_n}\left(\Delta\rho\,\vec{g} + \nabla P_c(S)\right) \\ &= \vec{H}_f(S) + \vec{H}_g(S) + \vec{H}_c(S).\end{aligned}$$



Summing up, we have:

$$H_{ik} = \frac{\lambda_w^u}{\lambda_w^u + \lambda_n^u} v_{ik} + \frac{\lambda_w^u \lambda_n^u}{\lambda_w^u + \lambda_n^u}\left[g_{ik} + P_{ik}\right]$$

where λ_w^u and λ_n^u 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: $S^{n+1} = S^n - \mathcal{F}(S^n, S^n)$. Implicit scheme: $\mathcal{F}(S^{n+1}, S^n) = 0$

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

$$H_{ik}(s) = \frac{\lambda_w^u(s_i, s_k)}{\lambda_w^u(s_i, s_k) + \lambda_n^u(s_i, s_k)} \big[v_{ik} + \lambda_n^u(s_i, s_k)(g_{ik} + P_{ik}) \big]$$

Implementation in MRST

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

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

$$H_{ik}(s) = \frac{\lambda_w^u(s_i, s_k)}{\lambda_w^u(s_i, s_k) + \lambda_n^u(s_i, s_k)} \big[v_{ik} + \lambda_n^u(s_i, s_k)(g_{ik} + P_{ik}) \big]$$

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] = twophaseJacobian(G, state, rock, fluid, 'pn1', pv1, ... )
```

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: $\mathcal{F}(S^{n+1}, S^n) = 0$

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

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

Explicit transport solver:

```
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
```

Implementation in MRST

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

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

$$H_{ik}(s) = \frac{\lambda_w^u(s_i, s_k)}{\lambda_w^u(s_i, s_k) + \lambda_n^u(s_i, s_k)} \big[v_{ik} + \lambda_n^u(s_i, s_k)(g_{ik} + P_{ik}) \big]$$

Implicit solver uses a Newton method:

$$\mathbf{0} = \boldsymbol{F}(\boldsymbol{s}_0 + \boldsymbol{\delta s}) \approx \boldsymbol{F}(\boldsymbol{s}_0) + \boldsymbol{J}(\boldsymbol{s}_0)\boldsymbol{\delta s},$$

$$\boldsymbol{J}(\boldsymbol{s}^\ell)\, \delta \boldsymbol{s}^{\ell+1} = -\boldsymbol{F}(\boldsymbol{s}^\ell), \qquad \boldsymbol{s}^{\ell+1} \leftarrow \boldsymbol{s}^\ell + \delta \boldsymbol{s}^{\ell+1}$$

Implementation in MRST

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

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

$$H_{ik}(s) = \frac{\lambda_w^u(s_i, s_k)}{\lambda_w^u(s_i, s_k) + \lambda_n^u(s_i, s_k)} \big[v_{ik} + \lambda_n^u(s_i, s_k)(g_{ik} + P_{ik}) \big]$$

Implicit solver uses a Newton method:

$$\mathbf{0} = \boldsymbol{F}(\boldsymbol{s}_0 + \boldsymbol{\delta s}) \approx \boldsymbol{F}(\boldsymbol{s}_0) + \boldsymbol{J}(\boldsymbol{s}_0)\boldsymbol{\delta s},$$

$$\boldsymbol{J}(\boldsymbol{s}^\ell)\, \delta \boldsymbol{s}^{\ell+1} = -\boldsymbol{F}(\boldsymbol{s}^\ell), \qquad \boldsymbol{s}^{\ell+1} \leftarrow \boldsymbol{s}^\ell + \delta \boldsymbol{s}^{\ell+1}$$

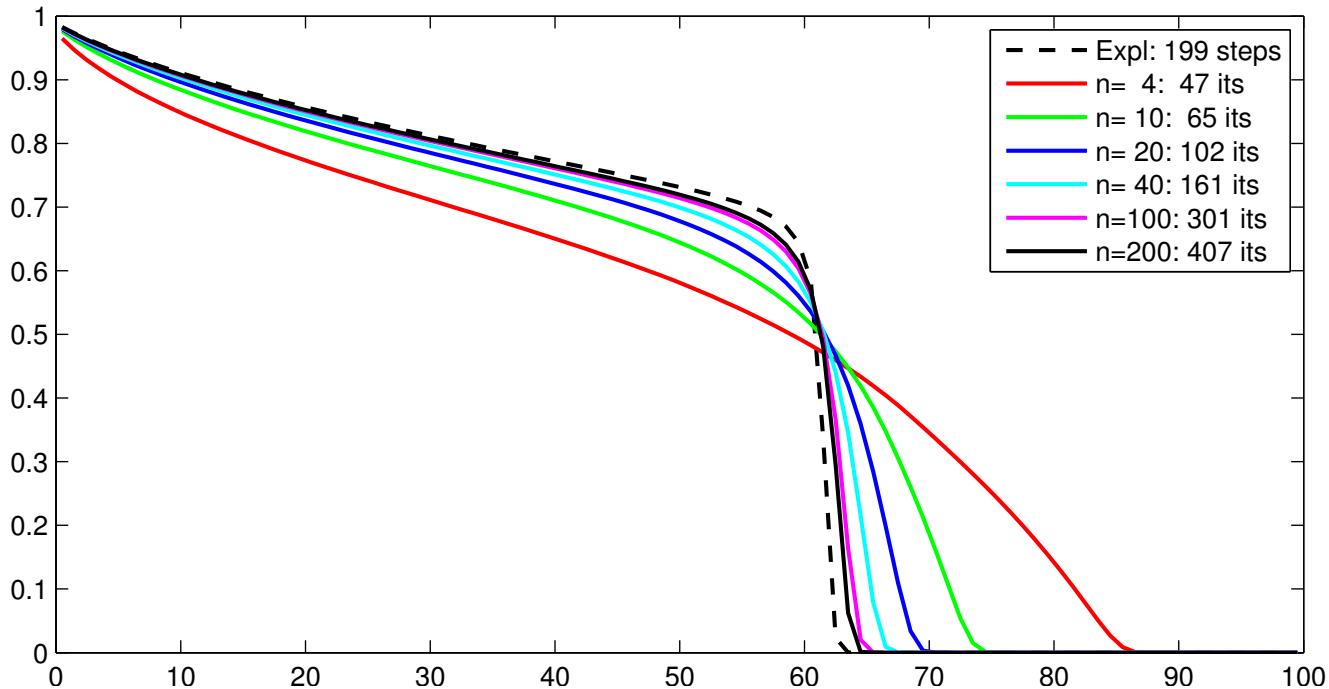
To get saturation values in $[0, 1]$, we need to introduce a *line-search* method that uses $\boldsymbol{p}^\ell = \delta \boldsymbol{s}^{\ell+1}$ as search direction. MRST uses an *inexact* method that asks for a sufficient decrease in $F(\boldsymbol{s}^\ell + \alpha\, \boldsymbol{p}^\ell)$ and reduces α in a geometric sequence.

Time-step control

```
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
   resSol = sn;
end
```

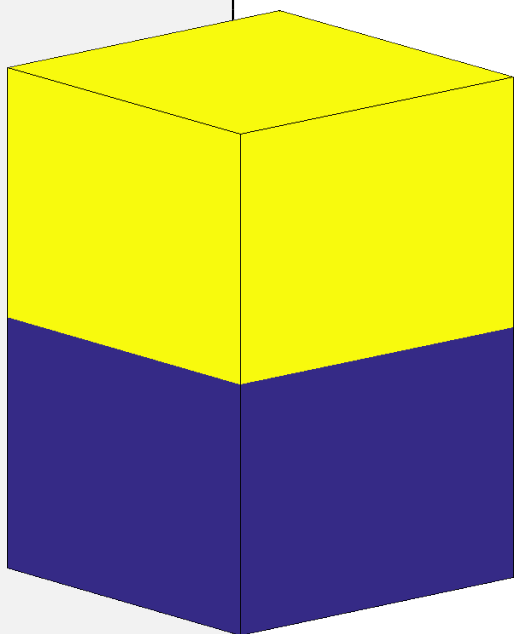
Example: Buckley–Leverett displacement

```
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);
```

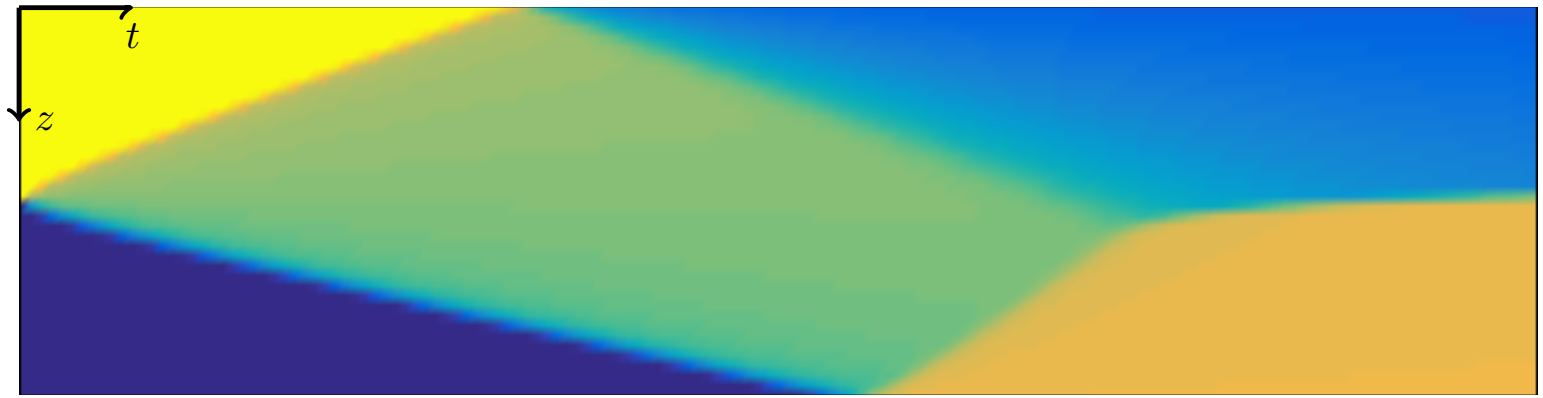


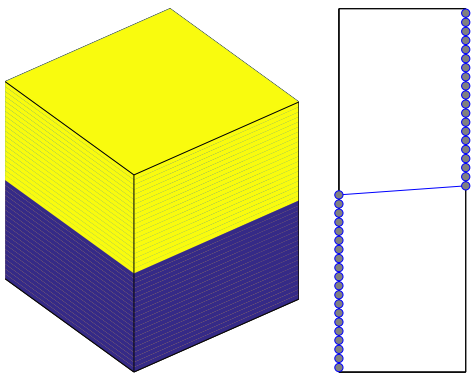
Example: inverted gravity column

```
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
```

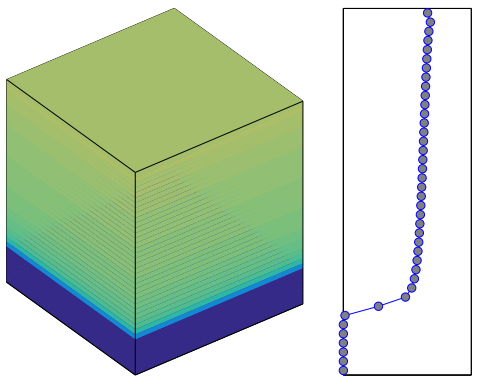


Example: inverted gravity column

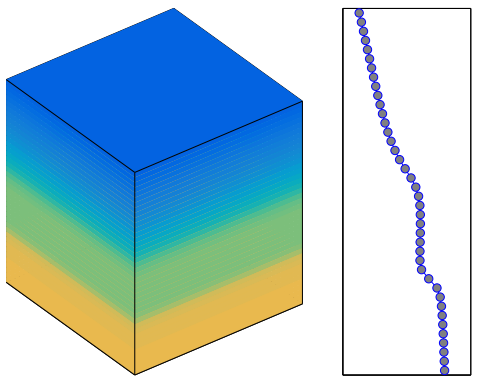




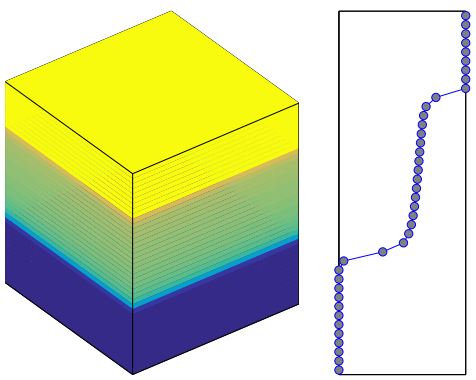
$t = 0$ days



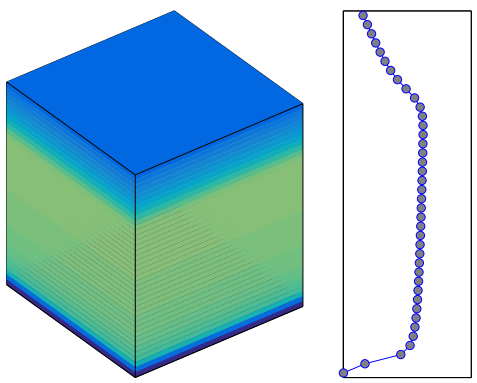
$t = 250$ days



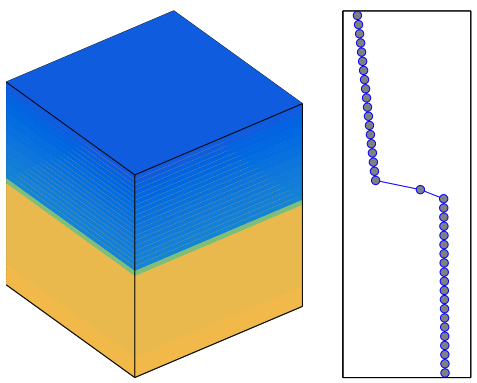
$t = 500$ days



$t = 125$ days

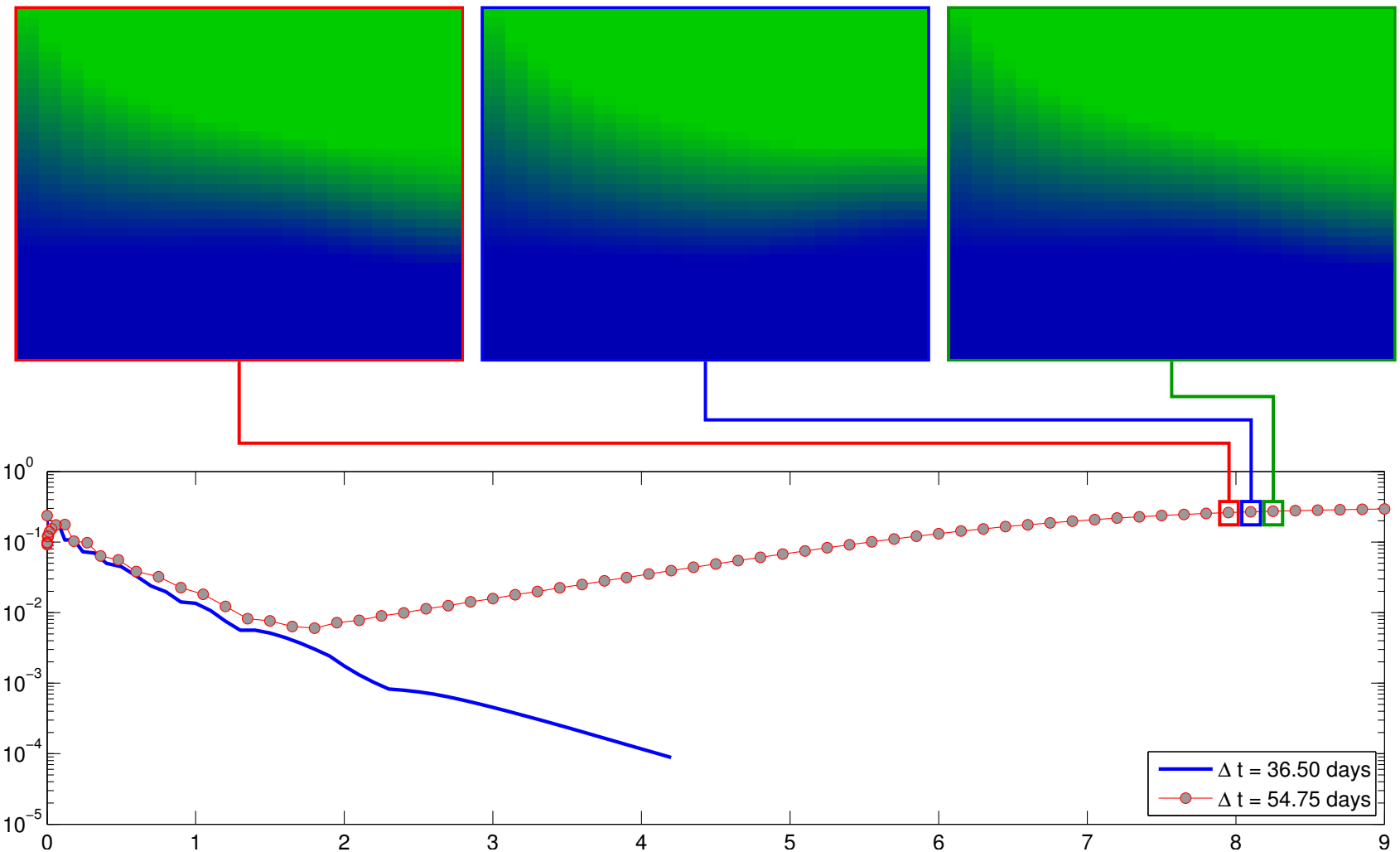


$t = 375$ days



$t = 750$ days

Potential pitfall: capillary-dominated flow



Outline

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

Single-phase weakly compressible flow

The governing equation is

$$c\frac{\partial p}{\partial t} - \nabla \cdot ((\mathbf{K}/\mu)\nabla p) = 0$$

Semi-discrete flow equations on residual form with implicit time discretization and discrete operators `div`, `grad`.

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

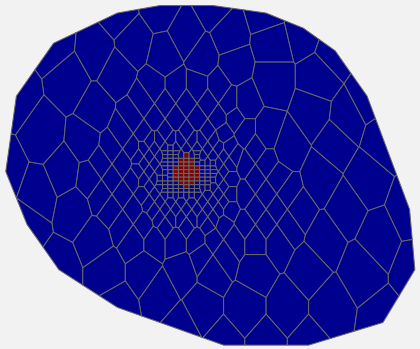
```
presEq = @(p, p0, dt)     (1/dt)*c*(p-p0) - div( (T/mu).*grad(p));
```

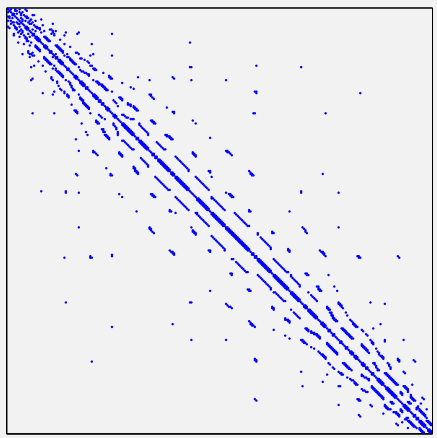
current time step

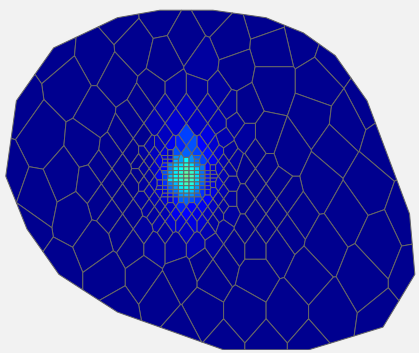
previous time step

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:

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

```
%  Fluid  properties
c = 1e-4;
mu = 1*centi*poise;
```

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

Model with rock and fluid compressibility:

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

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

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

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

avg is a face-average operator: $\mathbb{R}^{nc} \rightarrow \mathbb{R}^{nf}$

Adding effects: gravity

Semi-discrete flow equations on residual form:

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

Homogeneous equation implemented in MRST

```
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:

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

Homogeneous equation implemented in MRST

```
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));
```

current time step

previous time step

ρ at cell face

Adding effects: well model and controls

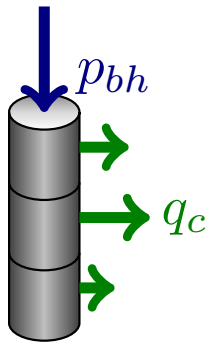
Peacemann well model, with hydrostatic pressure in well bore, and control on bottom-hole pressure:

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

$$q_c = \frac{\rho}{\mu}\mathrm{WI}(p_c - p),$$

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

$$p_{bh} = \text{constant}$$



Implemented in MRST:

```
wc = W(1).cells;   % connection grid cells
WI = W(1).WI;      % well-indices
dz = W(1).dZ;      % connection depth relative to bottom-hole

p_conn = @(bhp)        bhp + g*dz.*rho(bhp);
q_conn = @(p, bhp)     WI.*(rho(p(wc))/mu).*(p_conn(bhp) - p(wc));
rateEq = @(p, bhp, qS) qS-sum(q_conn(p, bhp))/rhoS;
ctrlEq = @(bhp)        bhp-100*barsa;
```

Details of simulator: time loop and assembly of equations

```
[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

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

    % Newton loop
    resNorm = 1e99;
    p0  = double(p); %
    nit = 0;
    while (resNorm > t
        % -- 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;
    end

    if nit > maxits,
        error('Newton s
    end
end
```

Adding effects: pressure-dependent viscosity

Assume the following model:

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

Adding effects: pressure-dependent viscosity

Assume the following model:

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

Arithmetic averaging:

```
mu = @(p) mu0*(1+c_mu*(p-pr));
v  = @(p)  -(T./mu(avg(p))).*(grad(p) - g*avg(rho(p)).*dz);

qcon = @(p,bhp) WI.*(rho(p(wc))./mu(p(wc))).*(pcon(bhp)-p(wc));
```

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] = getCellNoFaces(G);
hf2f = sparse(F,(1:numel(cn))',1);
hf2f = hf2f(all(C~=0,2),:);
fmob = @(mu,p) 1./( hf2f*(mu(p(cn))./hT) )

v = @(p) -fmob(mu,p).*(grad(p) - g*avg(rho(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 `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

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

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

Constitutive laws and operators

```
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

```
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) );

hEq = @(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));
```

Adding effects: thermal flow

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

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

Constitutive laws and operators

```
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

```
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) );

hEq = @(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)

$$\frac{(\boldsymbol{\phi S}_\alpha \boldsymbol{\rho}_\alpha)^{n+1} - (\boldsymbol{\phi S}_\alpha \boldsymbol{\rho}_\alpha)^n}{\Delta t^n} + \mathtt{div}(\boldsymbol{\rho v})_\alpha^{n+1} = (\boldsymbol{\rho q})_\alpha^{n+1}$$

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

Adding effects: multiple phases (without mass transfer)

$$\frac{(\boldsymbol{\phi S}_\alpha \boldsymbol{\rho}_\alpha)^{n+1} - (\boldsymbol{\phi S}_\alpha \boldsymbol{\rho}_\alpha)^n}{\Delta t^n} + \mathtt{div}(\boldsymbol{\rho v})_\alpha^{n+1} = (\boldsymbol{\rho q})_\alpha^{n+1}$$

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

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)./muO;
```

Adding effects: multiple phases (without mass transfer)

$$\frac{(\phi \boldsymbol{S}_\alpha \boldsymbol{\rho}_\alpha)^{n+1} - (\phi \boldsymbol{S}_\alpha \boldsymbol{\rho}_\alpha)^n}{\Delta t^n} + \mathtt{div}(\boldsymbol{\rho v})_\alpha^{n+1} = (\boldsymbol{\rho q})_\alpha^{n+1}$$

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

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

```
dp  = grad(p);
dpW = dp-g*avg(rW).*gradz;
dpO = dp-g*avg(rO).*gradz;
```

and use this to define upwind-weighted fluxes:

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

vW  = -upw(double(dpW) <= 0, rW.*mobW).*T.*dpW;
vO  = -upw(double(dpO) <= 0, rO.*mobO).*T.*dpO;
```

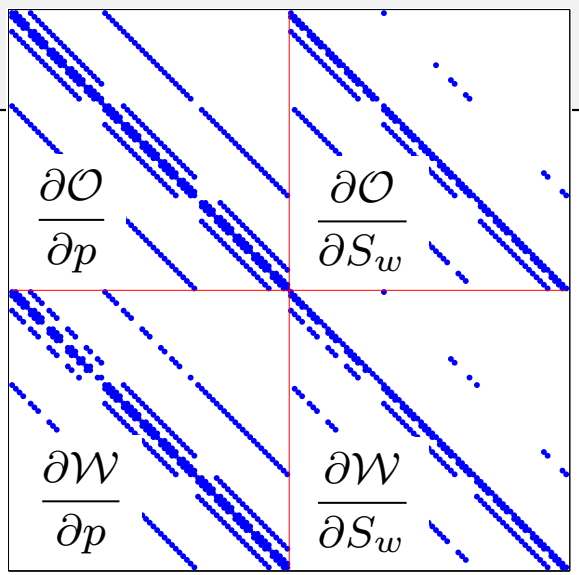
Adding effects: multiple phases (without mass transfer)

$$\frac{(\boldsymbol{\phi S}_\alpha \boldsymbol{\rho}_\alpha)^{n+1} - (\boldsymbol{\phi S}_\alpha \boldsymbol{\rho}_\alpha)^n}{\Delta t^n} + \mathtt{div}(\boldsymbol{\rho v})_\alpha^{n+1} = (\boldsymbol{\rho q})_\alpha^{n+1}$$

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

Now, we have all we need to compute residual equations

```
water = (1/dt(n)).*(vol.*rW.*sW - vol0.*rW0.*sW0) + div(vW);
oil   = (1/dt(n)).*(vol.*rO.*(1-sW) - vol0.*rO0.*(1-sW0)) + div(vO);
eqs   = {oil, water};
eq    = cat(eqs{:});
```



Adding effects: multiple phases (without mass transfer)

$$\frac{(\boldsymbol{\phi S}_\alpha \boldsymbol{\rho}_\alpha)^{n+1} - (\boldsymbol{\phi S}_\alpha \boldsymbol{\rho}_\alpha)^n}{\Delta t^n} + \mathtt{div}(\boldsymbol{\rho v})_\alpha^{n+1} = (\boldsymbol{\rho q})_\alpha^{n+1}$$

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

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

Outline

Advanced simulators: motivation

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

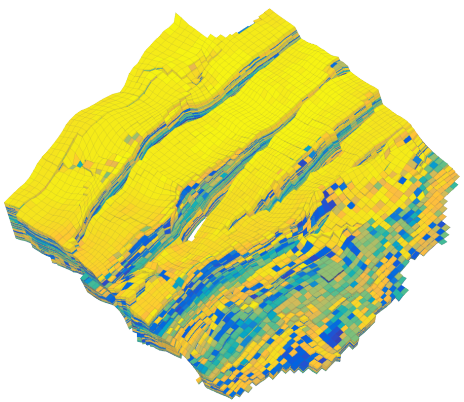
However, there are some disadvantages as well:

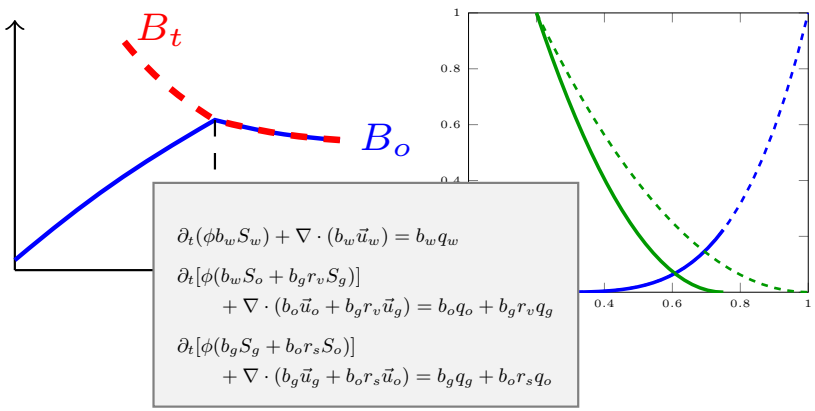
- mixing logic of Newton solver with definition of model equations
- time-stepping and plotting will be done per script
- implementing several variations of the same model will result in code duplication

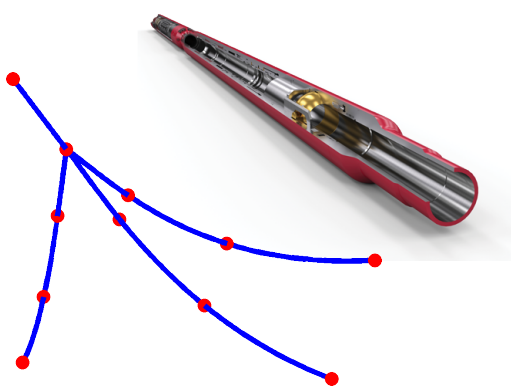
Advanced simulators: motivation

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

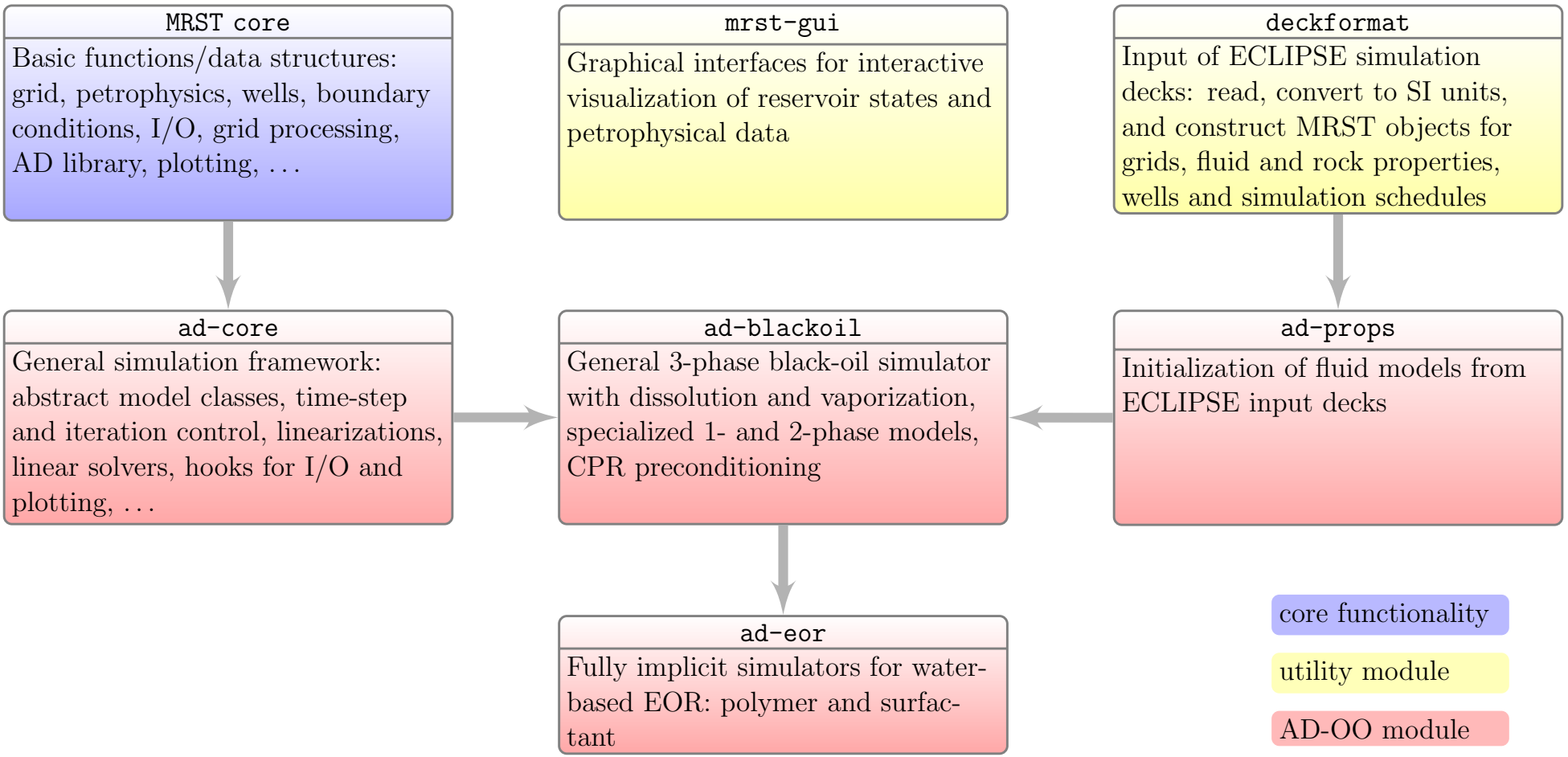
Next step: object-orientation

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 $\rightarrow$ migrate to class-based solver

The AD-OO modules



Example: two-phase Buckley–Leverett

```
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

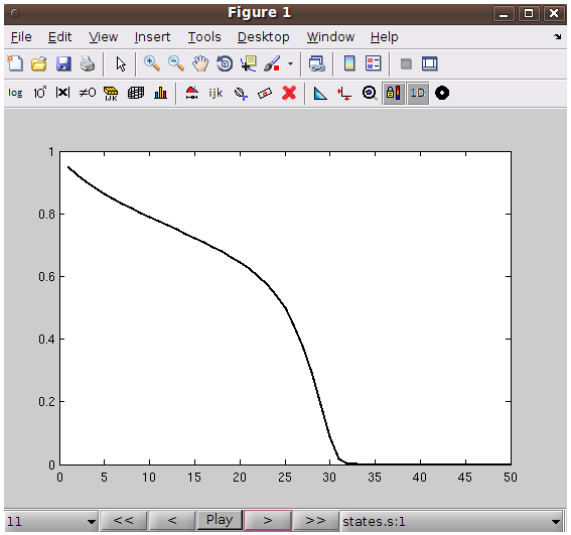
```
G     = cartGrid([50, 1, 1], [1000, 10, 10]*meter);
G     = computeGeometry(G);
rock  = makeRock(1*darcy*ones, .3);
fluid = initSimpleADIFluid('phases', 'WO', 'n', [2 2]);

% 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:

```
[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

```
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]);
```

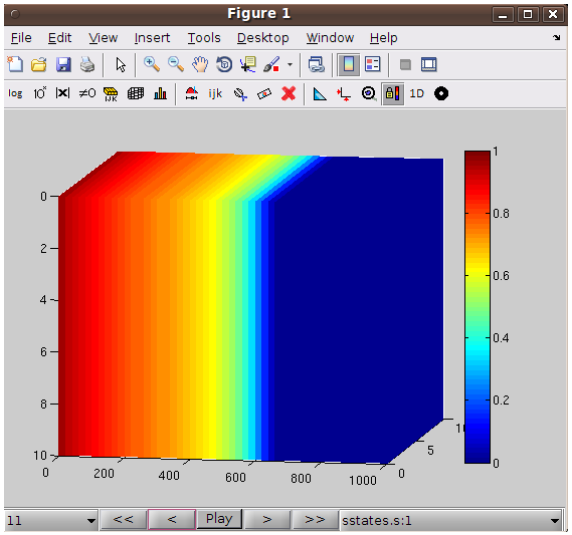


Simulate 1 PVI using a manual loop:

```
[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

```
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]);
```

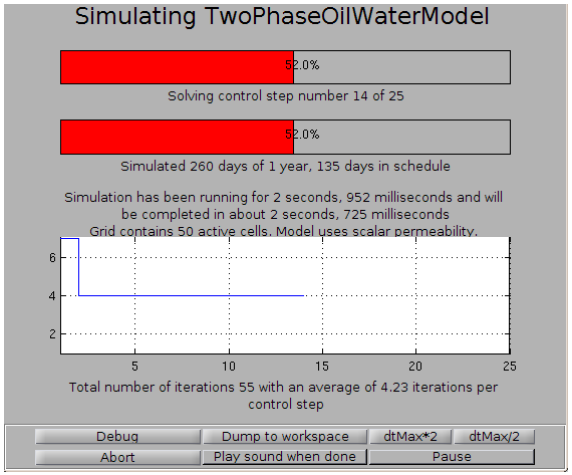


Repeat simulation with general solver:

```
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

```
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:

```
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

Classic Newton

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

Main loop

```
[x, t] = initializeSolution(...)
while t < T
   [ΔT, ctrl]=getControl(t)
   τ = 0
   while τ < ΔT
      do
         [Δt, ...]=getTimeStep(...)
         [ok, τ, ...]=solveMinistep(t + τ, Δt, ...)
      while ok=false
   end
end
```

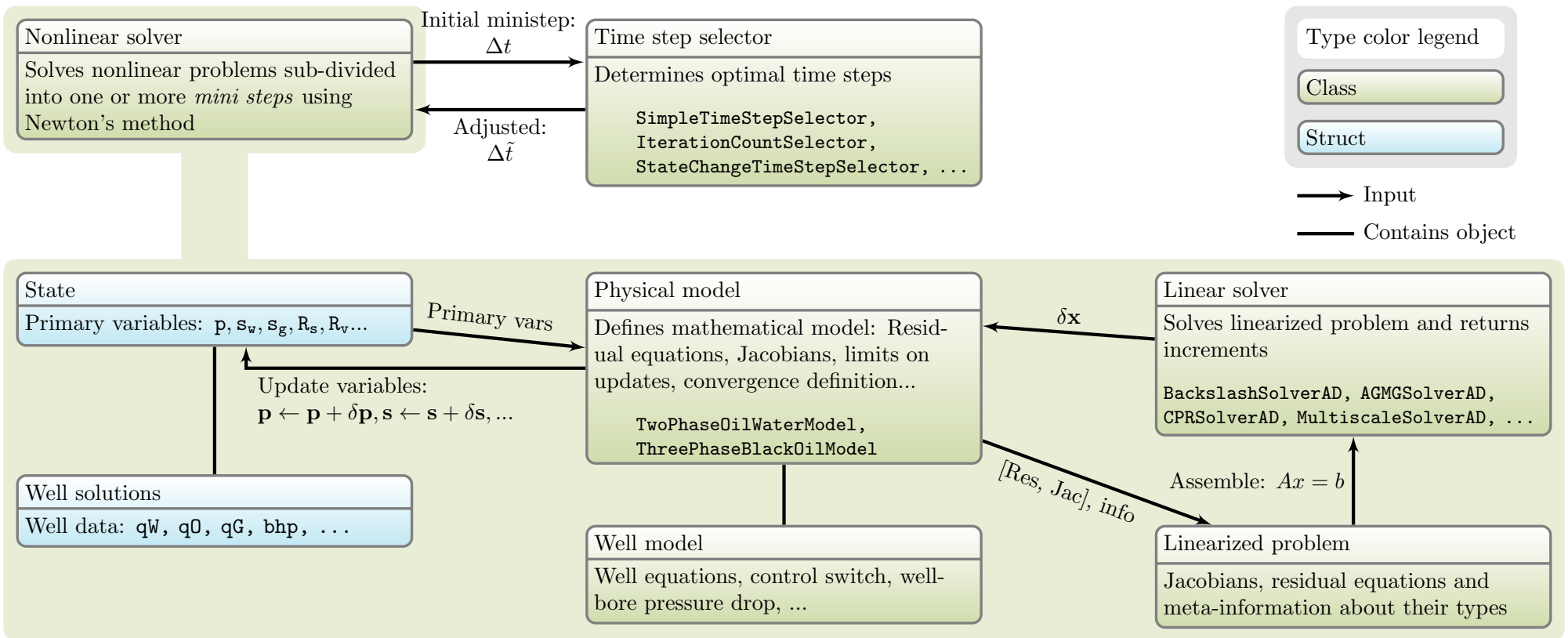
Context:

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

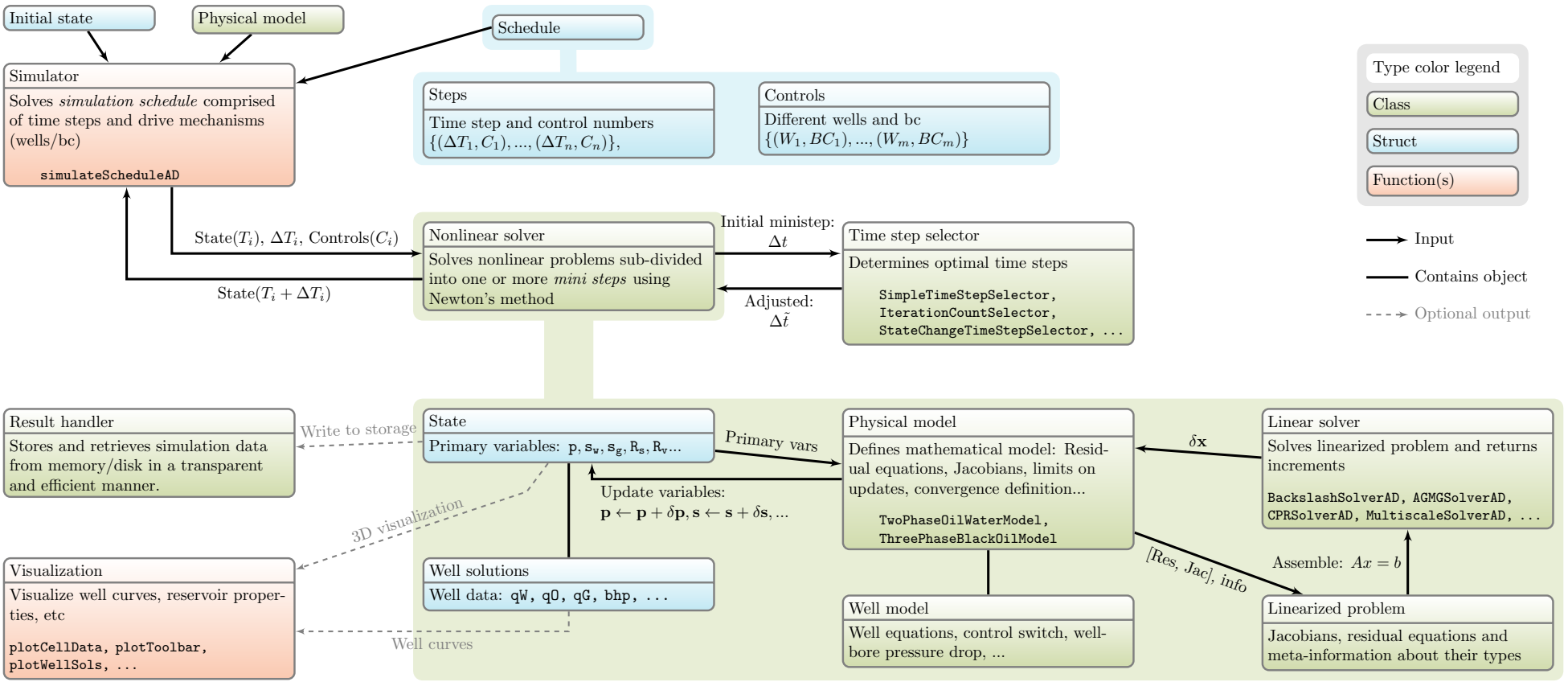
solveMinistep

```
[res, J, ...] = getEqs(t + τ, ...)
xit = x
while res > tol & it ≤ itmax
    lsys = assembleLinSys(res, J, ...)
    lsol = setupLinSolver(xit, lsys, ...)
    upd = solveLinSys(xit, lsys, lsol, ...)
    upd = stabilizeStep(xit, upd, lsys, ...)
    xit = updateIterate(upd, ...)
    cleanupLinSolver(lsol)
    [res, J ] = getEqs(t + τ, ...)
end
if it ≤ itmax
    ok = true
    [τ, x, ...] = updateSolution(xit)
else
    ok = false
end
```

More about the nonlinear solver

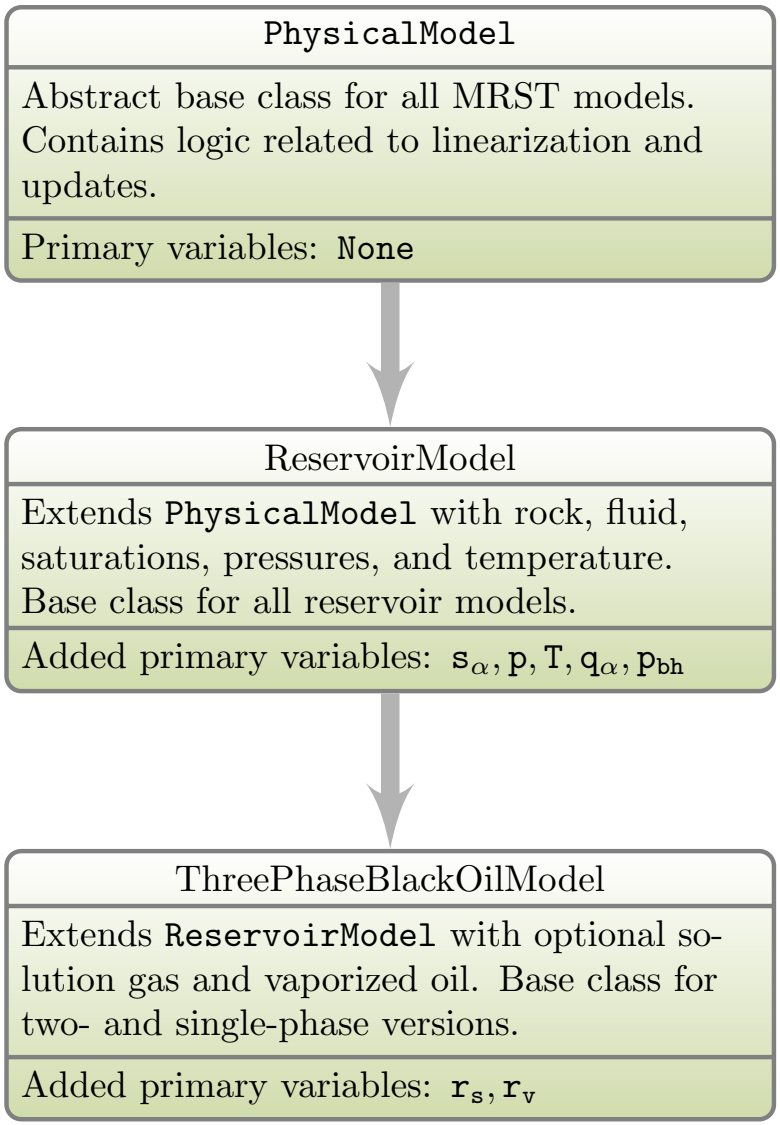


The layout of the AD solvers

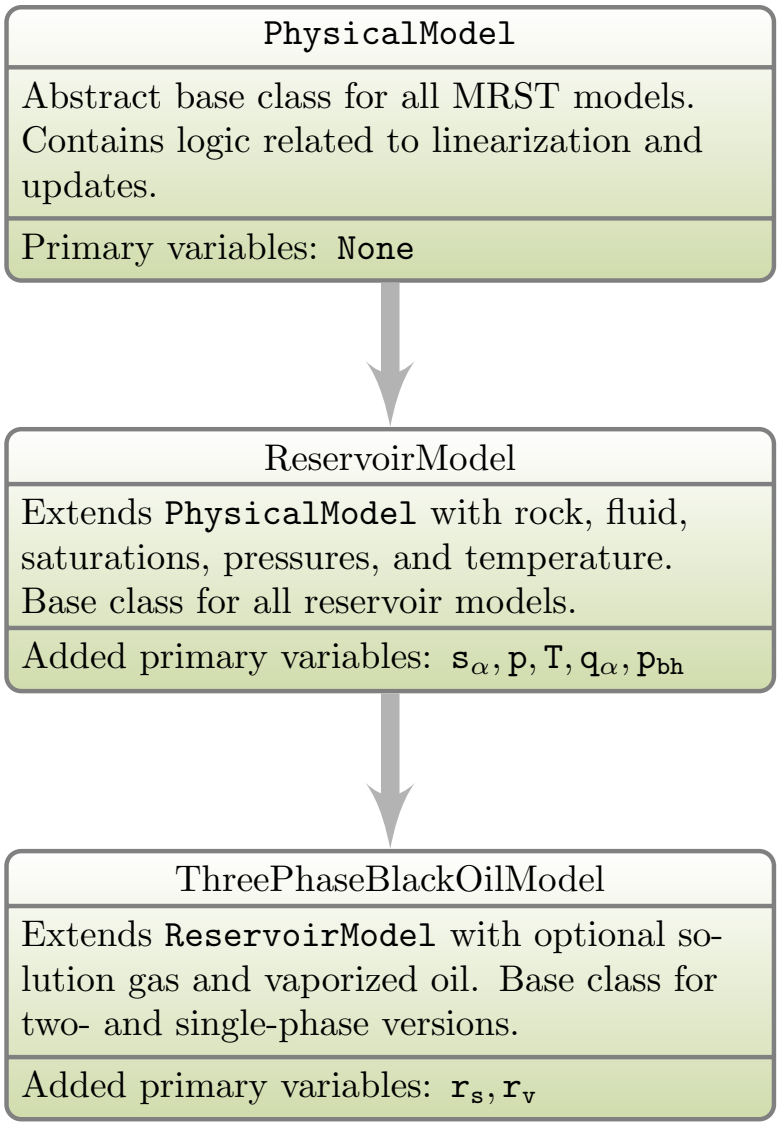


The framework is designed so that you can only work on the components you are interested in: If you want to write a flow solver, you do not need to debug a Newton solver.

Functionality through inheritance



Functionality through inheritance

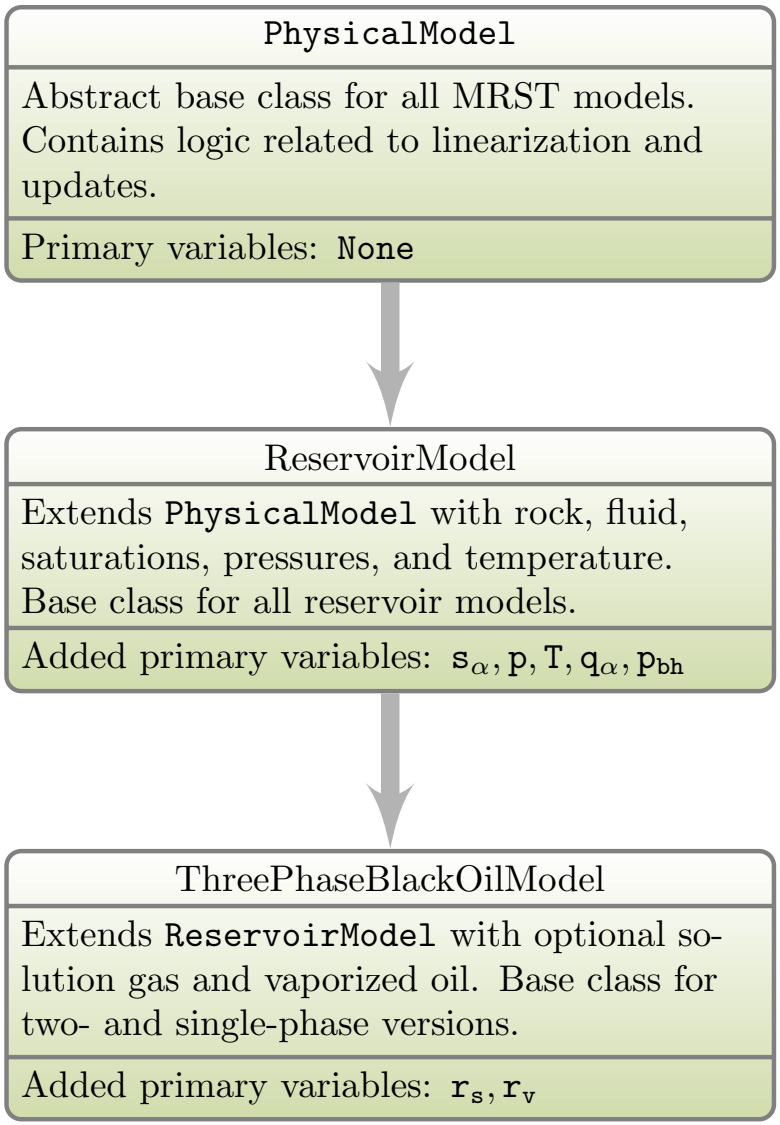


PhysicalModel

Properties:

```
operators, G
nonlinearTolerance, stepFunctionIsLinear
verbose
```

Functionality through inheritance



PhysicalModel

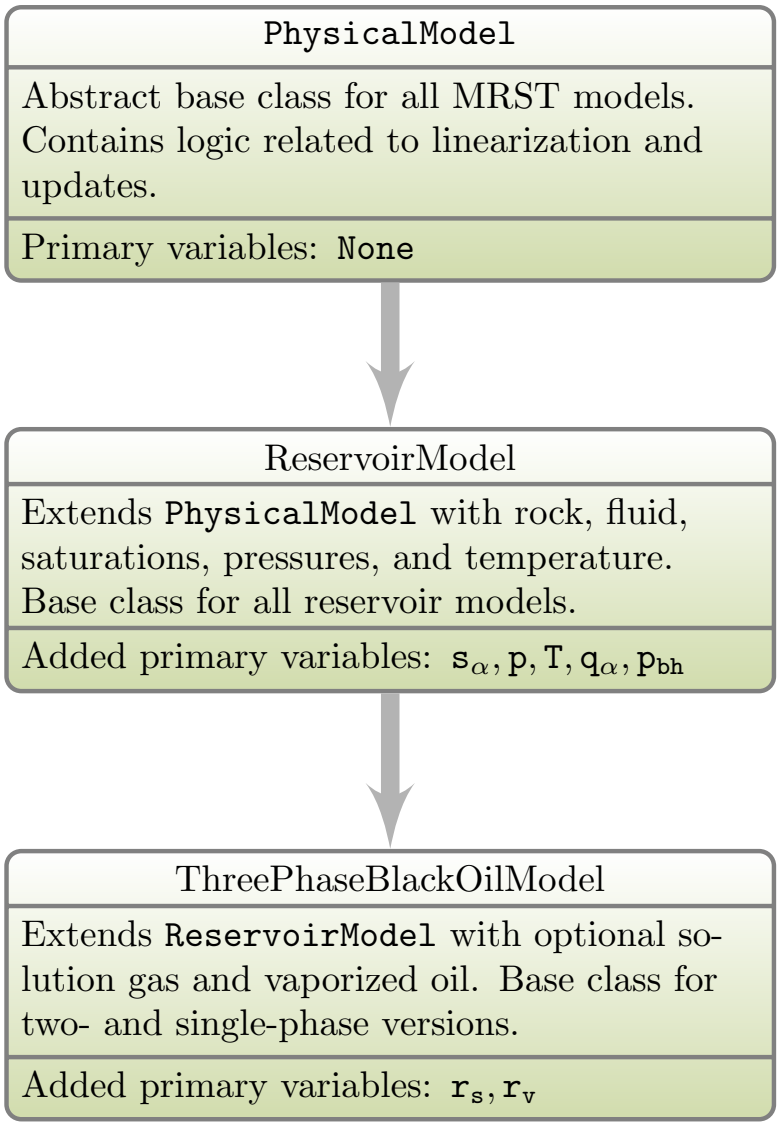
Properties:

```
operators, G
nonlinearTolerance, stepFunctionIsLinear
verbose
```

Quality assurance:

```
state = model.validateState(state)
model = model.validateModel(...)
```

Functionality through inheritance



PhysicalModel

Properties:

```
operators, G
nonlinearTolerance, stepFunctionIsLinear
verbose
```

Quality assurance:

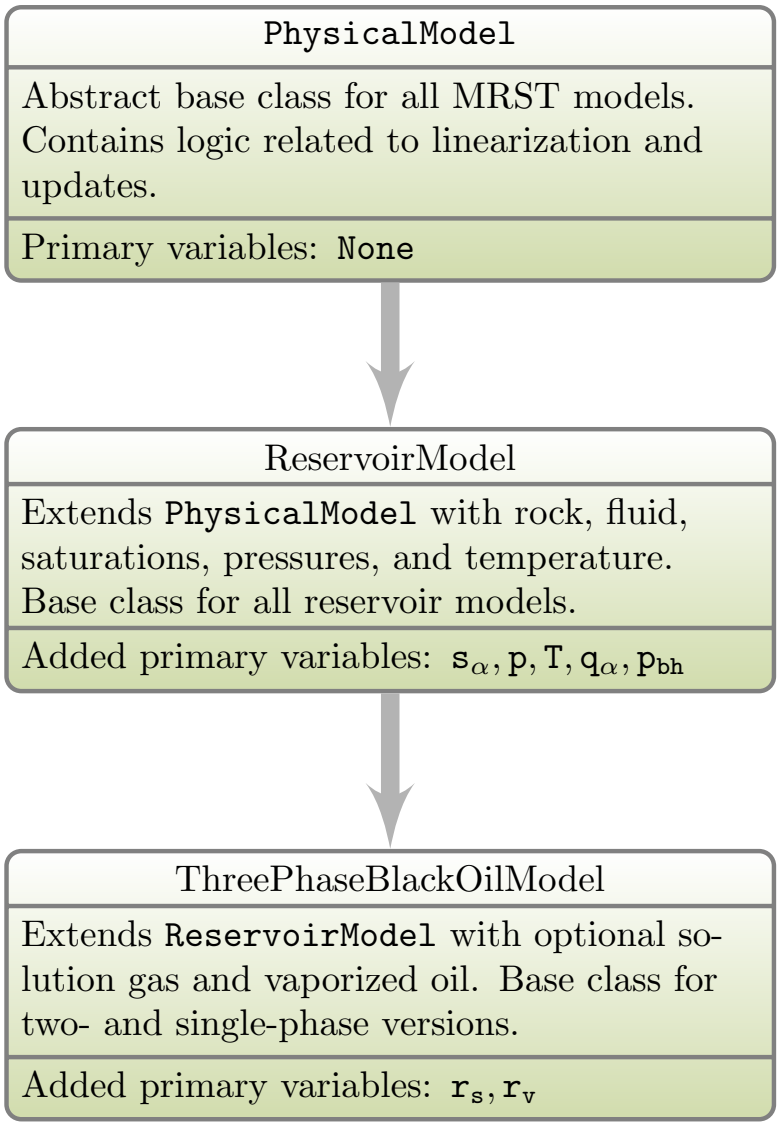
```
state = model.validateState(state)
model = model.validateModel(...)
```

Querying / setting model properties:

```
p     = model.getProp(state, 'pressure')
[p,s] = model.getProps(state, 'pressure', 's')
[f,i] = model.getVariableField(name)
state = model.setProp(model, state, 'pressure', 5)
state = model.incrementProp(state, 'pressure', 1)
state = model.capProperty(state,'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

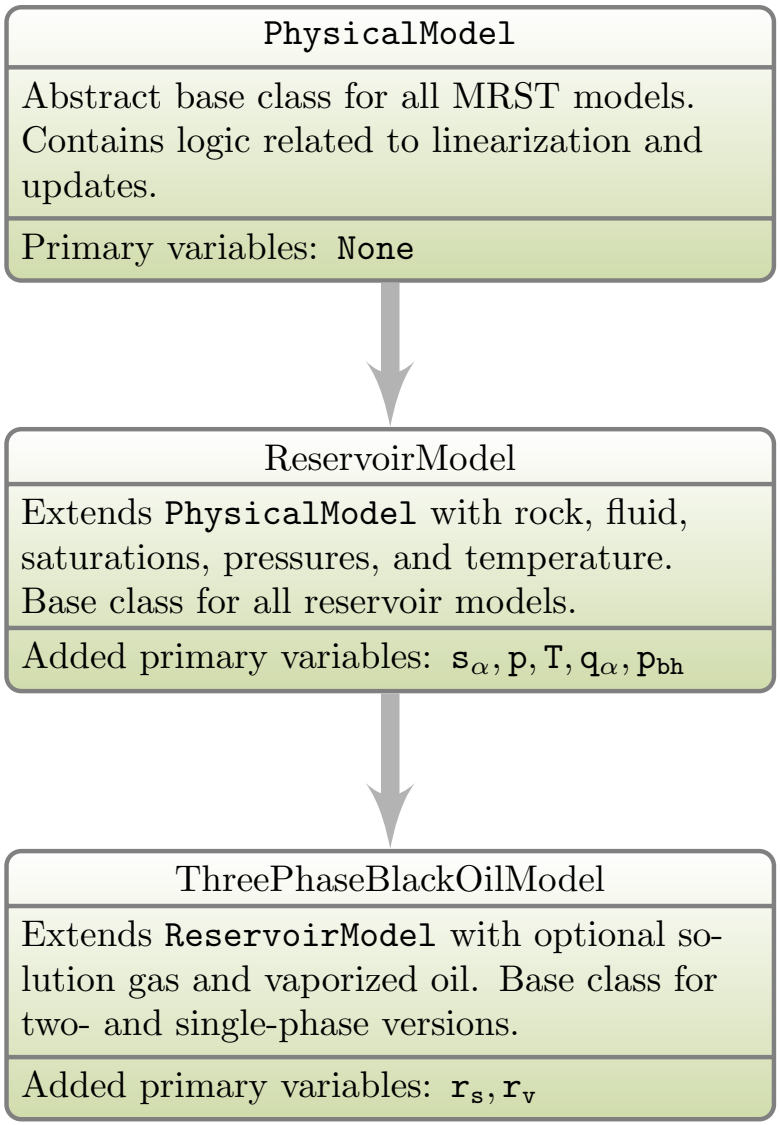


PhysicalModel

Get drive mechanisms:

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

Functionality through inheritance



PhysicalModel

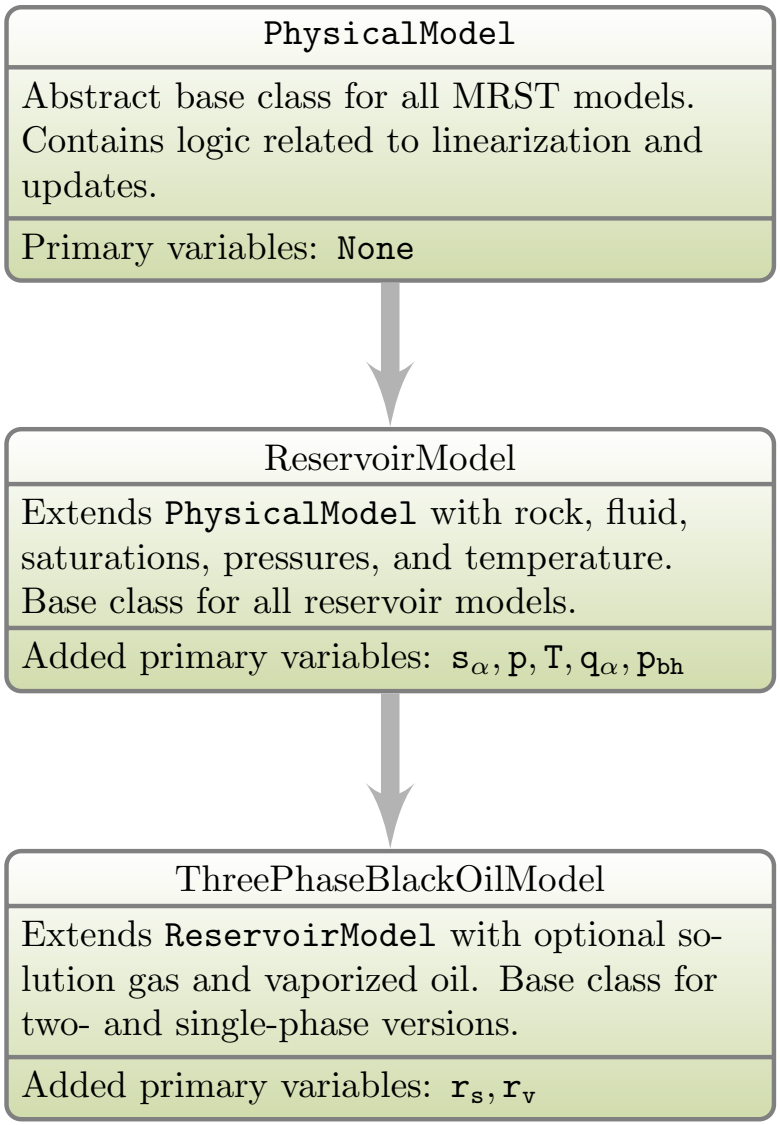
Get drive mechanisms:

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

Linearize and assemble discrete problem:

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

Functionality through inheritance



PhysicalModel

Get drive mechanisms:

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

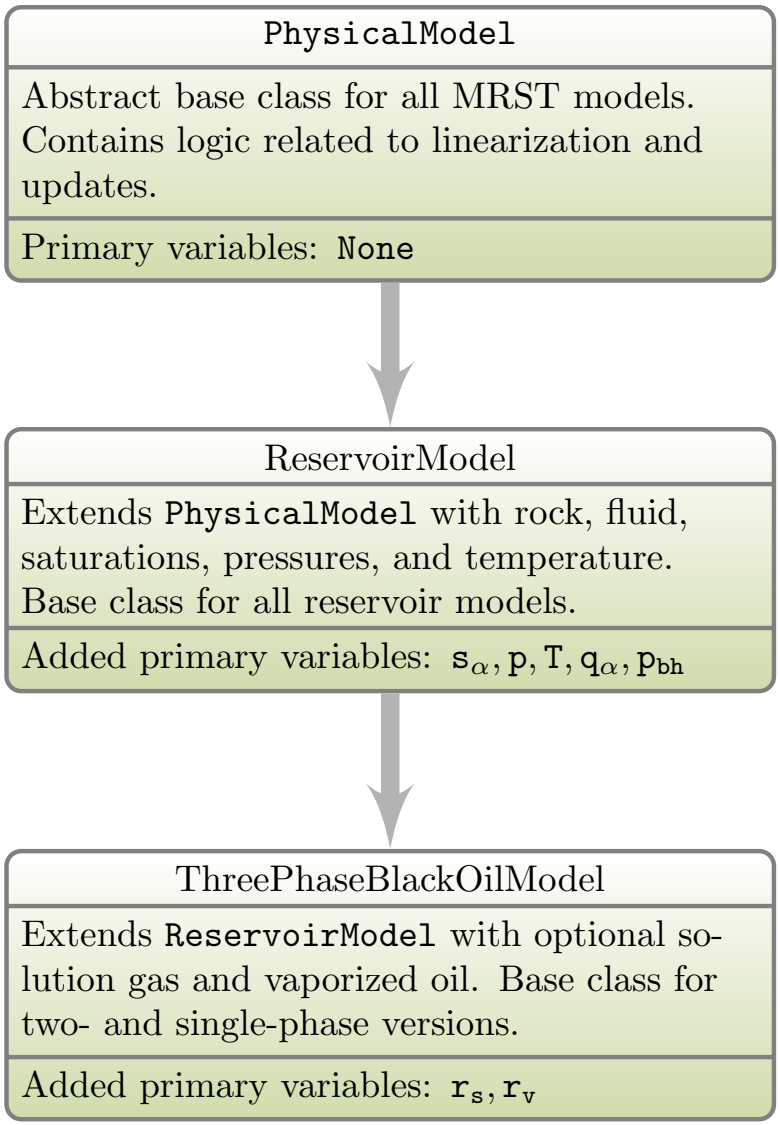
Linearize and assemble discrete problem:

```
[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



PhysicalModel

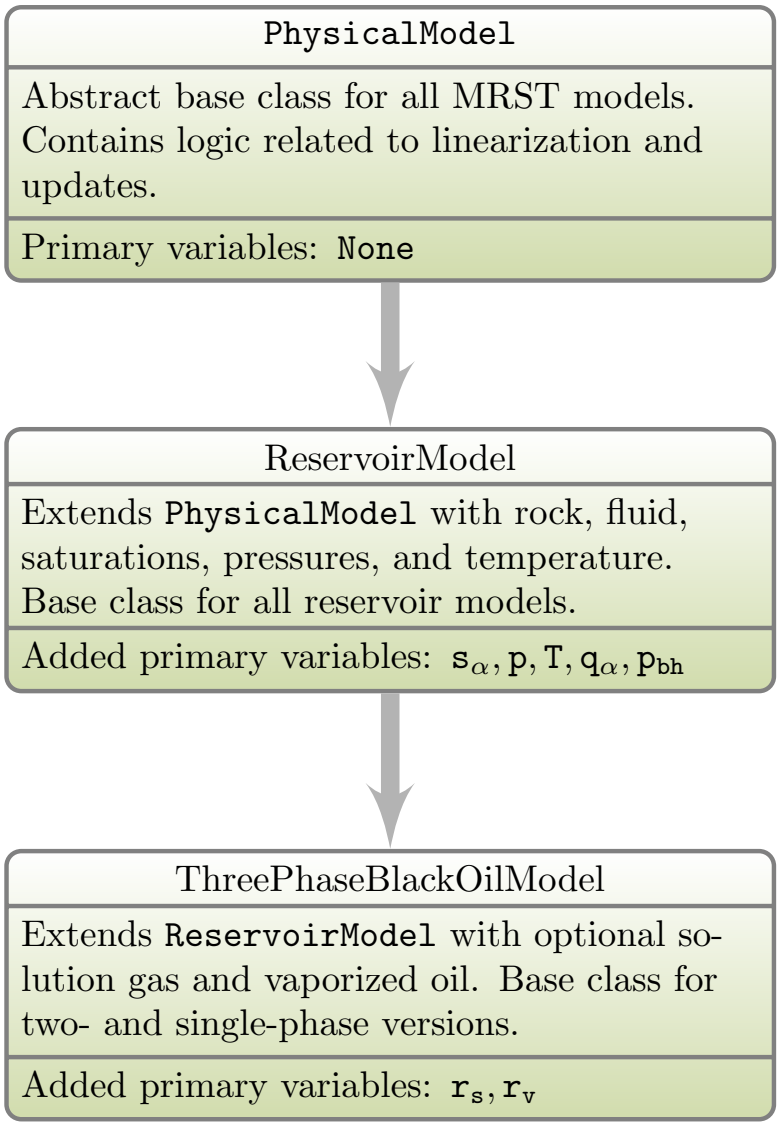
Update state from Newton increment:

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

and other utility functions:

```
[conv, ..] = model.checkConvergence(problem, n)

[state,rep] = model.updateAferConvergence(...
                state0, state, dt, drivingForces)
:
:
```

Functionality through inheritance

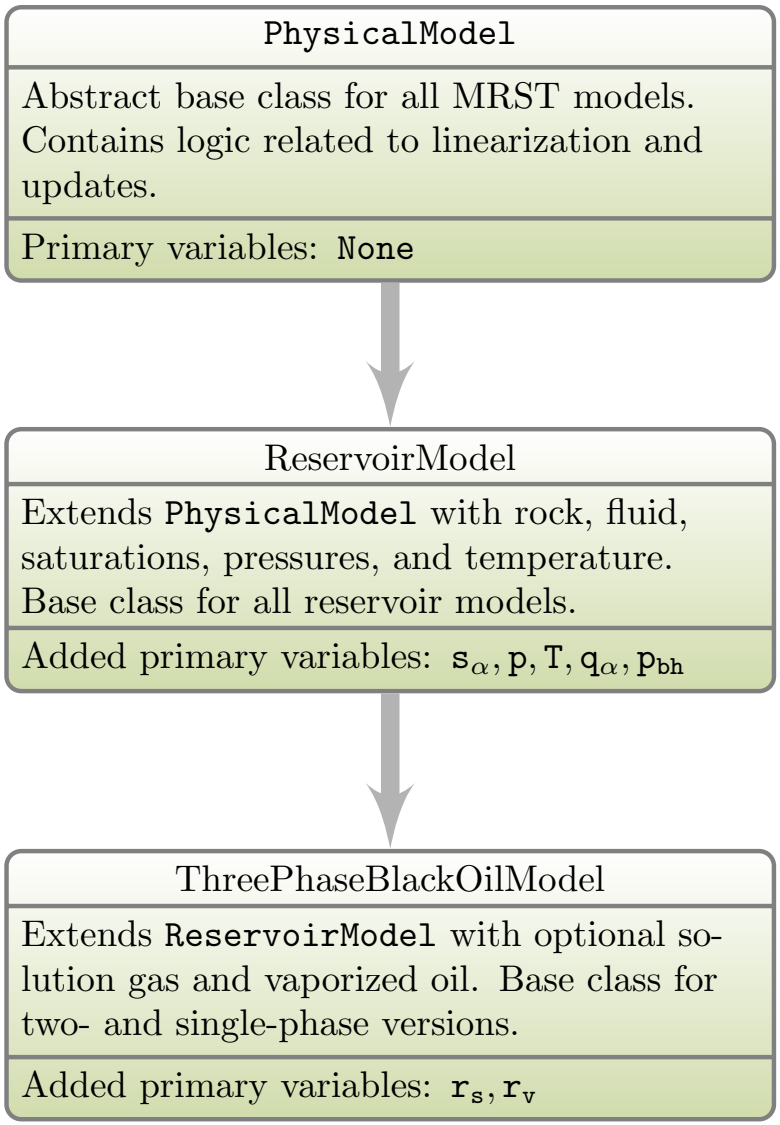


ReservoirModel

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



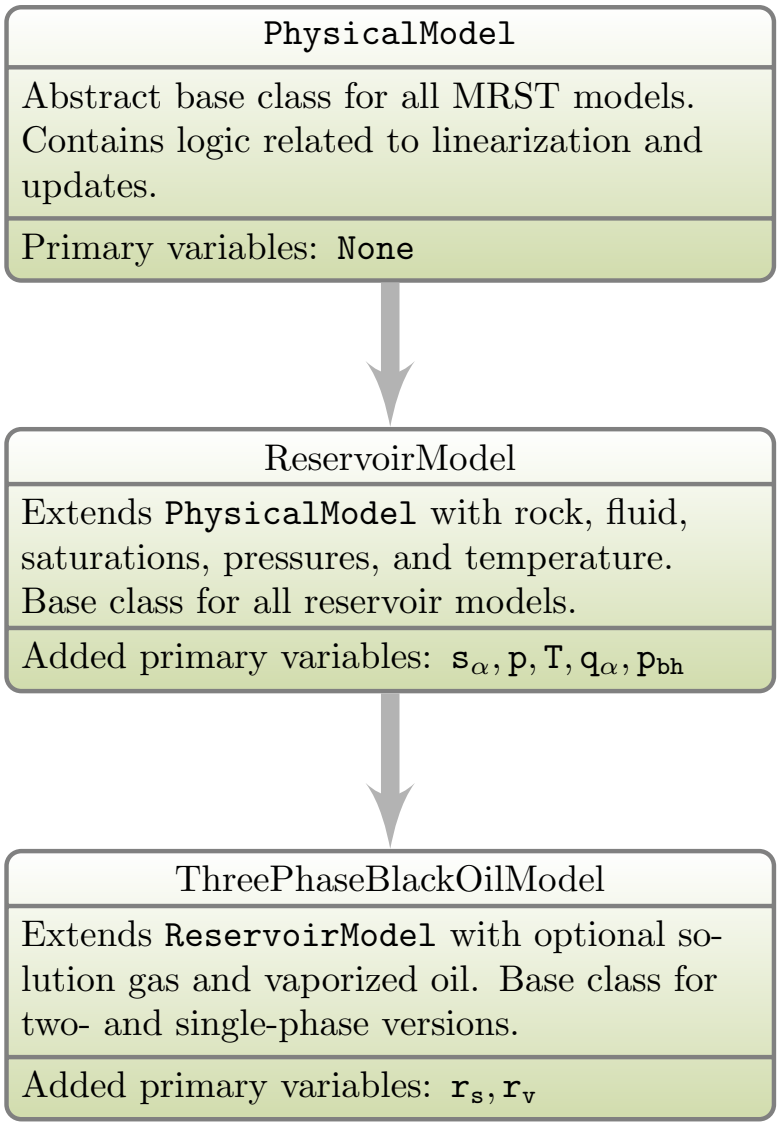
ReservoirModel

Declaration of physical variables:

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



ReservoirModel

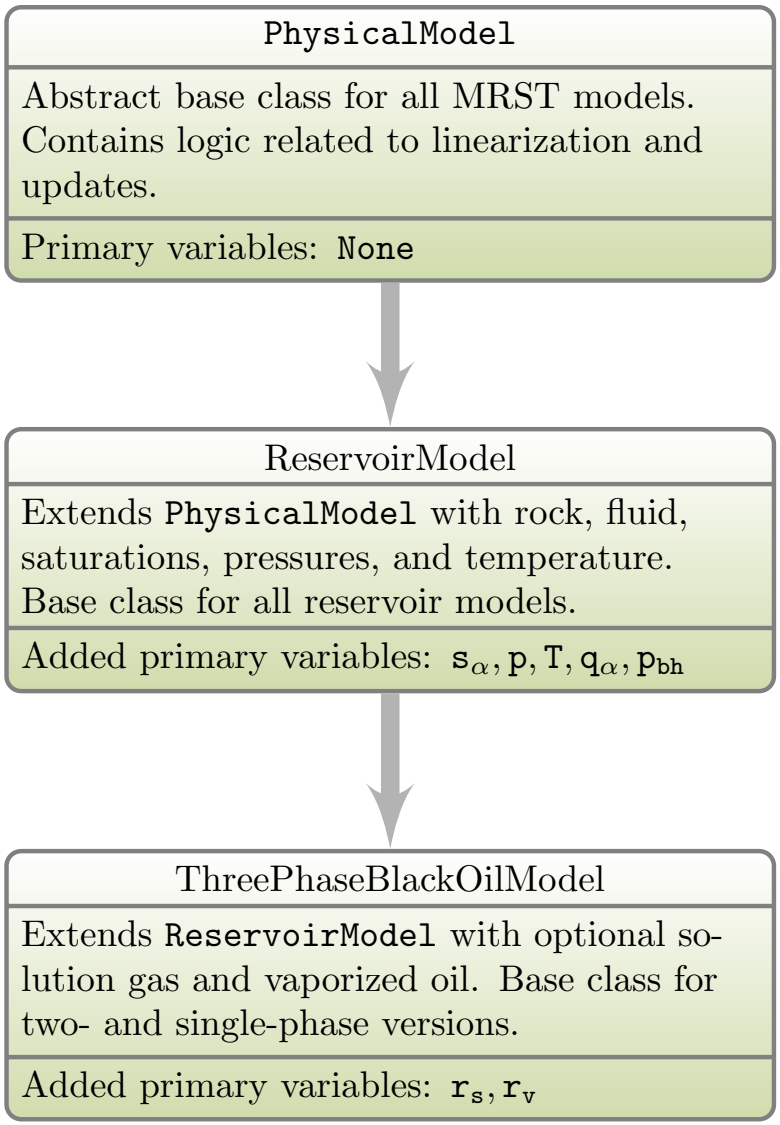
The class declares known drive mechanisms:

```
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

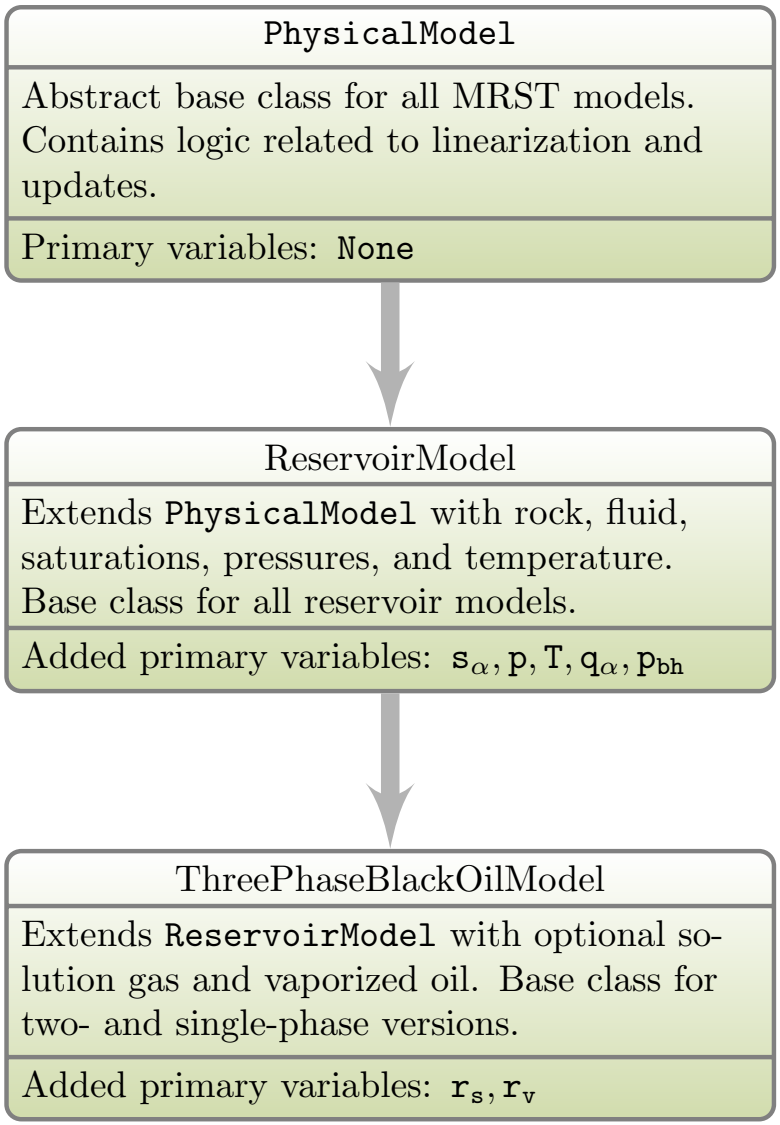


ReservoirModel

Default discretization is a two-point method:

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

Functionality through inheritance



ThreePhaseBlackOilModel

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

Evaluation of residual equations:

```
[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

```
DENSITY
                0282 1000      0.01 /
-- nb 721 is the reservoir density.  this needs the equivalent
-- surface density which is 0282 (i.e. 721*1.5 (FVF)
--

--
--       P           Bo         Vis
PVDO
100 1.5225 1.0
200 1.5075 1.0
250 1.5 1.0
400 1.4775 1.0 /
--       P           Bw         Cw         Vis     Viscosibility
PVTW
250 1.0 1e-5 0.4 0.0 /
--
--       P             Cr
ROCK
        250 2e-5 /
-- INCLUDE
```

Input parser

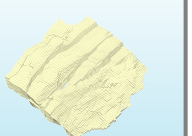
Reads complete simulation decks: grid and petrophysics, fluid and rock properties, region information, well definitions, operating schedule, convergence control, etc

Data
Class
Struct
Function
Input
Contains

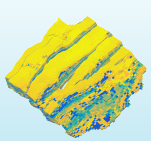
Reservoir model

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

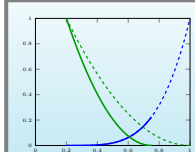
- **Grid**



- **Petrophysics**



- **Fluids**



State

Physical variables inside the reservoir

$\mathtt{p}, \mathtt{s}_\mathtt{w}, \mathtt{s}_\mathtt{o}, \mathtt{s}_\mathtt{g}, \mathtt{c}, \mathtt{r}_\mathtt{v}, \mathtt{r}_\mathtt{s}$

Well state

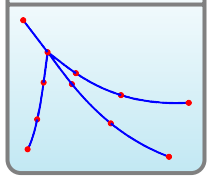
Physical variables inside the wellbore

$\mathtt{q}^\mathtt{s}_\mathtt{w}, \mathtt{q}^\mathtt{s}_\mathtt{o}, \mathtt{q}^\mathtt{s}_\mathtt{g}, \mathtt{q}^\mathtt{s}_\mathtt{p}, \mathtt{p}_\mathtt{bh}$

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

```
-- ==================================================================
-- THIS IS THE FIRST SPE COMPARISON PROBLEM,"COMPARISON OF SOLUTIONS TO A
-- THREE-DIMENSIONAL BLACK-OIL RESERVOIR SIMULATION PROBLEM", REPORTED
-- BY AZIS AND ODEH AT THE SPE SYMPOSIUM ON RESERVOIR SIMULATION ,
-- JANUARY 1981. IT IS  A NON SWELLING AND SWELLING STUDY. A REGULAR
-- GRID WITH TWO WELLS (INJECTOR AND PRODUCER)AND A IMPES SOLUTION METHOD
-- IS USED FOR THIS SIMULATION.THE PRODUCTION IS CONTROLLED BY FLOW RATE
-- AND MIN. BHP. OIL RATE, GOR, PRESSURE AND GAS SATURATION ARE TO BE REPO
-- ==================================================================
RUNSPEC
TITLE
   ODEH PROBLEM - IMPLICIT OPTION - 1200 DAYS

DIMENS
   10   10    3  /

NONNC

OIL

WATER

GAS

DISGAS

FIELD

EQLDIMS
    1  100   10    1    1 /

TABDIMS
    1    1   16   12    1   12 /

WELLDIMS
    2    1    1    2 /

NUPCOL
    4 /

START
  19 'OCT' 1982  /

NSTACK
   24 /

--FMTOUT

--FMTIN

UNIFOUT

UNIFIN

--NOSIM

--IMPES
```

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

```
GRID    ================================================================
-------- IN THIS SECTION , THE GEOMETRY OF THE  SIMULATION GRID AND THE
-------- ROCK PERMEABILITIES AND POROSITIES ARE DEFINED.
----------------------------------------------------------------------
--  THE X AND Y DIRECTION CELL SIZES ( DX, DY ) AND THE POROSITIES ARE
--  CONSTANT THROUGHOUT THE GRID. THESE ARE SET IN THE FIRST 3 LINES
--  AFTER THE EQUALS KEYWORD. THE CELL THICKNESSES ( DZ ) AND
--  PERMEABILITES ARE THEN SET FOR EACH  LAYER. THE CELL TOP DEPTHS
--  ( TOPS ) ARE NEEDED ONLY IN THE TOP LAYER ( THOUGH THEY COULD BE.
--  SET THROUGHOUT THE GRID ). THE SPECIFIED MULTZ VALUES ACT AS
--  MULTIPLIERS ON THE TRANSMISSIBILITIES BETWEEN THE CURRENT LAYER
--  AND THE LAYER BELOW.
INIT

INCLUDE
'./SPE1.GRDECL'
/

PORO
300*0.3
/

PERMX
100*500.0
100*50.0
100*200.0
/

PERMY
100*500.0
100*50.0
100*200.0
/

-- Note: ignoring  MULTZ!
--   layer 1-2    'MULTZ' 0.64      /
--   layer 2-3    'MULTZ' 0.265625  /
-- Reducing PERMZ a little instead
PERMZ
100*300.0
100*30.0
100*50.0
/
```

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

```
PROPS    ================================================================
-------- THE PROPS SECTION DEFINES THE REL. PERMEABILITIES, CAPILLARY
-------- PRESSURES, AND THE PVT PROPERTIES OF THE RESERVOIR FLUIDS
-----------------------------------------------------------------------
-- WATER RELATIVE PERMEABILITY AND CAPILLARY PRESSURE ARE TABULATED AS
-- A FUNCTION OF WATER SATURATION.
--
-- Generated with MRST's family_1() function from the original deck.
--  SWAT   KRW   KRO   PCOW
SWOF
   0.120000000000000                  0   1.000000000000000
   0.121000000000000   0.000000011363636   1.000000000000000
   0.140000000000000   0.000000227272727   0.997000000000000
   0.170000000000000   0.000000568181818   0.980000000000000
   0.240000000000000   0.000001363636364   0.700000000000000
   0.319999999000000   0.000002272727261   0.350000004375000
   0.370000000000000   0.000002840909091   0.200000000000000
   0.420000000000000   0.000003409090909   0.090000000000000
   0.520000000000000   0.000004545454545   0.021000000000000
   0.570000000000000   0.000005113636364   0.010000000000000
   0.620000000000000   0.000005681818182   0.001000000000000
   0.720000000000000   0.000006818181818   0.000100000000000
   0.820000000000000   0.000007954545455   0.000000000000000
   1.000000000000000   0.000010000000000                   0
/


SGOF
                  0                  0   1.000000000000000
   0.001000000000000                  0   1.000000000000000
   0.020000000000000                  0   0.997000000000000
   0.050000000000000   0.005000000000000   0.980000000000000
   0.120000000000000   0.025000000000000   0.700000000000000
   0.200000000000000   0.075000000000000   0.350000000000000
   0.250000000000000   0.125000000000000   0.200000000000000
   0.300000000000000   0.190000000000000   0.090000000000000
   0.400000000000000   0.410000000000000   0.021000000000000
   0.450000000000000   0.600000000000000   0.010000000000000
   0.500000000000000   0.720000000000000   0.001000000000000
   0.600000000000000   0.870000000000000   0.000100000000000
   0.700000000000000   0.940000000000000                   0
   0.850000000000000   0.980000000000000                   0
   0.880000000000000   0.984000000000000                   0
/

-- PVT PROPERTIES OF WATER
--
--    REF. PRES. REF. FVF  COMPRESSIBILITY  REF VISCOSITY  VISCOSIBILITY
PVTW
       4014.7     1.029        3.13E-6           0.31            0 /

-- ROCK COMPRESSIBILITY
--
--    REF. PRES   COMPRESSIBILITY
ROCK
        14.7          3.0E-6          /

-- SURFACE DENSITIES OF RESERVOIR FLUIDS
--
--        OIL   WATER   GAS
DENSITY
         49.1   64.79  0.06054  /

-- PVT PROPERTIES OF DRY GAS (NO VAPOURISED OIL)
-- WE WOULD USE PVTG TO SPECIFY THE PROPERTIES OF WET GAS
--
--   PGAS   BGAS   VISGAS
```

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

```
SOLUTION ====================================================================
-------- THE SOLUTION SECTION DEFINES THE INITIAL STATE OF THE SOLUTION
-------- VARIABLES (PHASE PRESSURES, SATURATIONS AND GAS-OIL RATIOS)
------------------------------------------------------------------------------
-- DATA FOR INITIALISING FLUIDS TO POTENTIAL EQUILIBRIUM
--
--    DATUM  DATUM   OWC    OWC    GOC    GOC    RSVD   RVVD   SOLN
--    DEPTH  PRESS  DEPTH   PCOW  DEPTH   PCOG  TABLE  TABLE   METH
-- EQUIL
--        8400   4800   8500    0     8200    0      1      0       0  /

-- VARIATION OF INITIAL RS WITH DEPTH
--
--    DEPTH    RS
-- RSVD
--        8200  1.270
--        8500  1.270  /

-- PRESSURE
-- 100*3.297832774859256e7
-- 100*3.302313357125603e7
-- 100*3.309483500720813e7
-- /
PRESSURE
100*4783.10205078125
100*4789.60058593750
100*4800.00000000000
/

SWAT
300*0.12
/

SGAS
300*0.0
/

-- RS
-- 300*226.1966570852417
-- /
RS
300*1.27
/

-- OUTPUT CONTROLS (SWITCH ON OUTPUT OF INITIAL GRID BLOCK PRESSURES)
RPTSOL
   1  11*0  /
```

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

```
SUMMARY  =====================================================================
-------- THIS SECTION SPECIFIES DATA TO BE WRITTEN TO THE SUMMARY FILES
-------- AND WHICH MAY LATER BE USED WITH THE ECLIPSE GRAPHICS PACKAGE
------------------------------------------------------------------------------

EXCEL

SEPARATE

--REQUEST PRINTED OUTPUT OF SUMMARY FILE DATA

RUNSUM

-- FIELD OIL PRODUCTION
FOPR

-- WELL GAS-OIL RATIO FOR PRODUCER
WGOR
-- 'PRODUCER'
/
-- WELL BOTTOM-HOLE PRESSURE
WBHP
-- 'PRODUCER'
/

-- GAS AND OIL SATURATIONS IN INJECTION AND PRODUCTION CELL
BGSAT
10 10 3 /
1  1  1 /
/
BOSAT
10 10 3 /
1  1  1 /
/

-- PRESSURE IN INJECTION AND PRODUCTION CELL
BPR
10 10 3 /
1  1  1 /
/
```

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

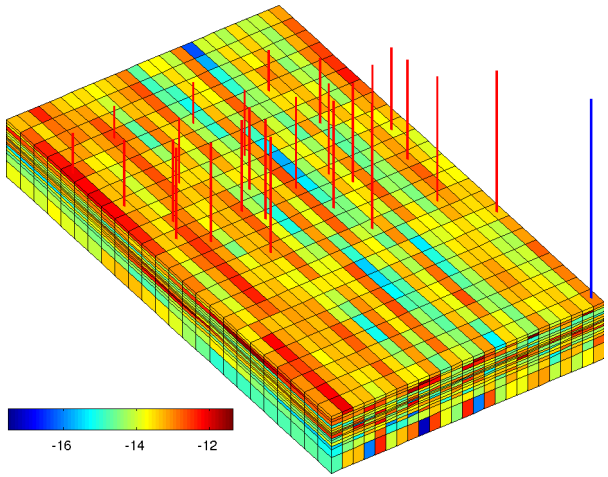
Data file: SPE1 benchmark

```
SCHEDULE ============================================================
-------- THE SCHEDULE SECTION DEFINES THE OPERATIONS TO BE SIMULATED
------------------------------------------------------------------------
-- CONTROLS ON OUTPUT AT EACH REPORT TIME
RPTSCHED
   1   0   1   1   0    0   4   2   2   0    0   2   0   0   0
   0   0   0   0   0    0   0   0   0   0    0   0   0   0   0
   0   0   0   0   0    0   0   0   1   0    0   0   0   0   0 /


--IMPES
-- 1.0 1.0 10000.0 /

-- SET 'NO RESOLUTION' OPTION
--DRSDT
-- 0 /

-- SET INITIAL TIME STEP TO 1 DAY AND MAXIMUM TO 6 MONTHS
TUNING
1  182.5  /
1.0 0.5 1.0E-6 /
/

-- WELL SPECIFICATION DATA
--
--     WELL   GROUP LOCATION  BHP   PI
--     NAME   NAME    I  J   DEPTH DEFN
WELSPECS
    'PRODUCER' 'G'   10 10    8400 'OIL'  /
    'INJECTOR' 'G'    1  1    8335 'GAS'  /
/

-- COMPLETION SPECIFICATION DATA
--
--     WELL     -LOCATION- OPEN/ SAT CONN  WELL
--     NAME     I  J K1 K2 SHUT  TAB FACT  DIAM
COMPDAT
    'PRODUCER'  10 10 3  3 'OPEN' 0   -1   0.5  /
    'INJECTOR'   1  1 1  1 'OPEN' 1   -1   0.5  /
  /

-- PRODUCTION WELL CONTROLS
--
--      WELL     OPEN/  CNTL   OIL  WATER   GAS  LIQU   RES   BHP
--      NAME     SHUT   MODE  RATE   RATE  RATE  RATE  RATE
WCONPROD
     'PRODUCER' 'OPEN' 'ORAT' 20000  4*                     1000 /
  /
-- WCONPROD
--      'PRODUCER' 'OPEN' 'BHP' 5*                     1000 /
--   /

-- INJECTION WELL CONTROLS
--
--      WELL     INJ   OPEN/  CNTL    FLOW
--      NAME    TYPE   SHUT   MODE    RATE
-- WCONINJ
--     'INJECTOR' 'GAS' 'OPEN' 'RATE'  100000  /
WCONINJE
     'INJECTOR' 'GAS' 'OPEN' 'RATE' 100000 100000 50000/
  /

--  YEAR 1

TSTEP
--0.2343    0.1393    0.1840    0.2189    0.2235
1.0 2*2.0 2*5.0 5*10.0 11*25.0
```

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

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

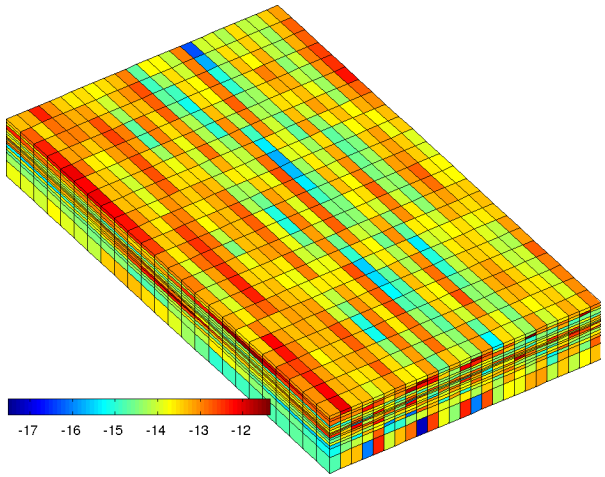


J. E. Killough (1995). Ninth SPE comparative solution project: A reexamination of black-oil simulation, doi: 10.2118/29110-MS

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

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9



Reading input and construct basic MRST objects:

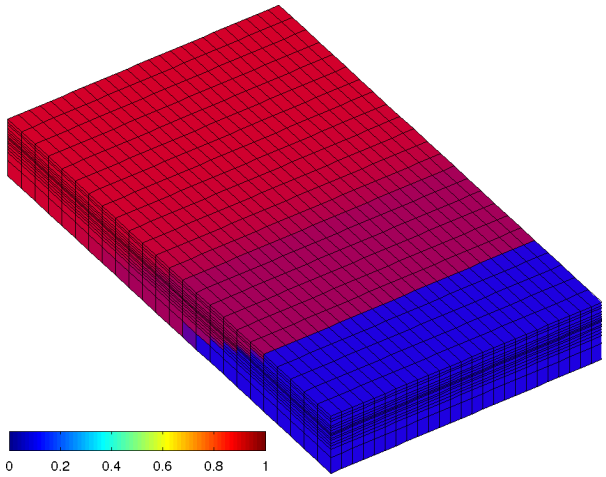
```
pth  = fullfile(getDatasetPath('spe9'), 'BENCH_SPE9.DATA');
deck = readEclipseDeck(fn);
deck = convertDeckUnits(deck);

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

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

fluid = initDeckADIFluid(deck);
```

J. E. Killough (1995). Ninth SPE comparative solution project: A reexamination of black-oil simulation, doi: 10.2118/29110-MS

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

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9



Initialization from given state in the input file:

```
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);
```

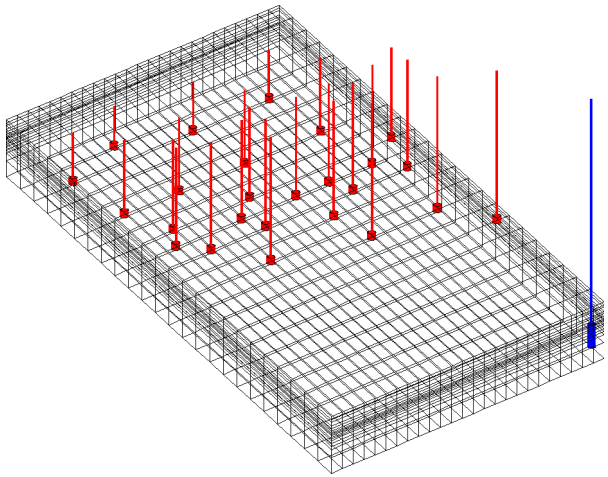
Generally, one may have to solve an equilibrium problem to set the initial state.

J. E. Killough (1995). Ninth SPE comparative solution project: A reexamination of black-oil simulation, doi: 10.2118/29110-MS

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

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9



Create model and simulation schedule:

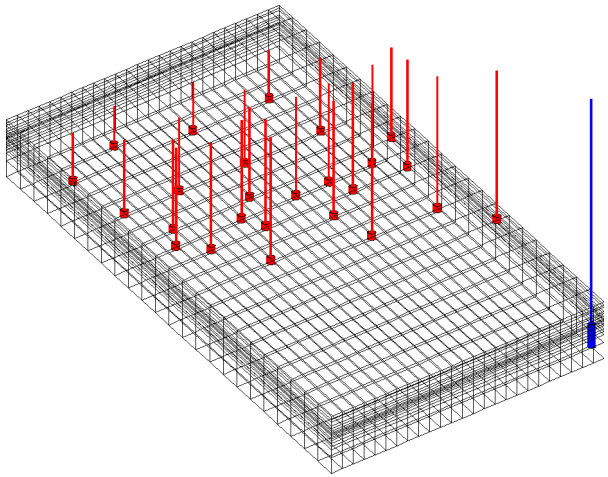
```
model = selectModelFromDeck(G, rock, fluid, deck);

% 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);
```

J. E. Killough (1995). Ninth SPE comparative solution project: A reexamination of black-oil simulation, doi: 10.2118/29110-MS

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

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);
```

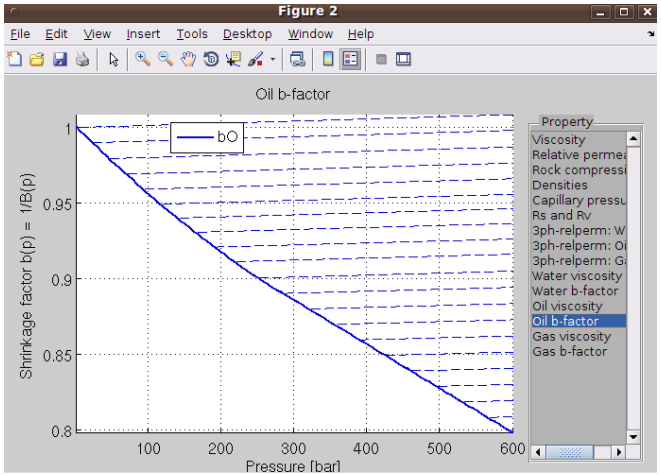
We select a CPR-type solver, with AGMG as multigrid preconditioner. The CPR preconditioner attempts to decouple the linear system into a pressure component and a transport component. Although not necessary here, it improves CPU time

J. E. Killough (1995). Ninth SPE comparative solution project: A reexamination of black-oil simulation, doi: 10.2118/29110-MS

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

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9



Inspect the rock-fluid and PVD properties:

```
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.

J. E. Killough (1995). Ninth SPE comparative solution project: A reexamination of black-oil simulation, doi: 10.2118/29110-MS

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

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9

```
Solving timestep 01/35:                                  -> 1 Day
Well INJE1: Control mode changed from rate to bhp.
================================================
| It # | CNV_W    | CNV_O    | CNV_G    | MB_W
================================================
|    1 | 2.82e-02 | 1.10e+00 | 5.00e-03 | 1.03e-05
|    2 | 1.14e-01 | 7.25e-02 | 8.13e-02 | 5.21e-06
Well PROD26: Control mode changed from orat to bhp
|    3 | 7.29e-03 | 1.29e-02 | 7.41e-02 | 3.40e-06
|    4 | 1.68e-03 | 3.79e-03 | 1.85e-02 | 7.87e-06
|    5 |*9.60e-04 | 6.20e-03 | 4.29e-03 | 1.64e-06
|    6 |*6.79e-04 | 1.45e-03 |*9.00e-04 | 1.19e-06
|    7 |*2.66e-04 |*8.31e-04 |*5.90e-04 | 2.56e-07
|    8 |*8.83e-05 |*3.48e-04 |*8.72e-05 |*4.58e-08
|    9 |*2.08e-05 |*7.35e-05 |*4.96e-05 |*1.05e-08
================================================
Solving timestep 02/35: 1 Day                            -> 2 Day
:
:
```

Run the schedule

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

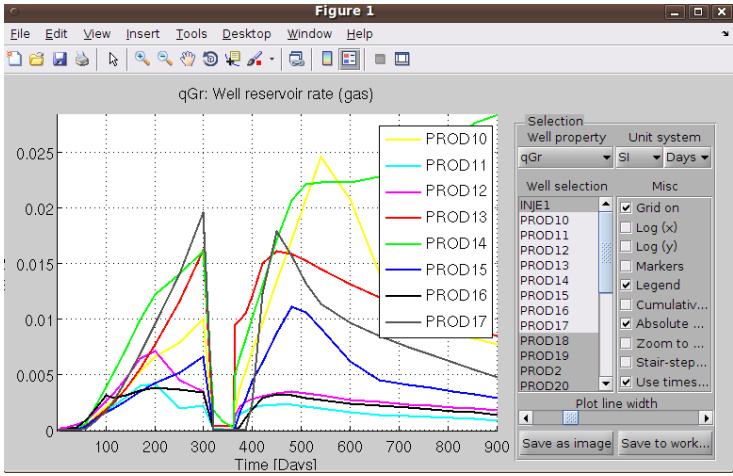
We give the schedule with well controls and control time steps. The simulator may use other timesteps internally, but it will always return values at the specified control steps. Setting `model.verbose=false` removes extensive reports about convergence, etc.

J. E. Killough (1995). Ninth SPE comparative solution project: A reexamination of black-oil simulation, doi: 10.2118/29110-MS

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

From: ad-blackoil/examples/spe9/blackOilTutorialSPE9



Launch a viewer to inspect reservoir responses:

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

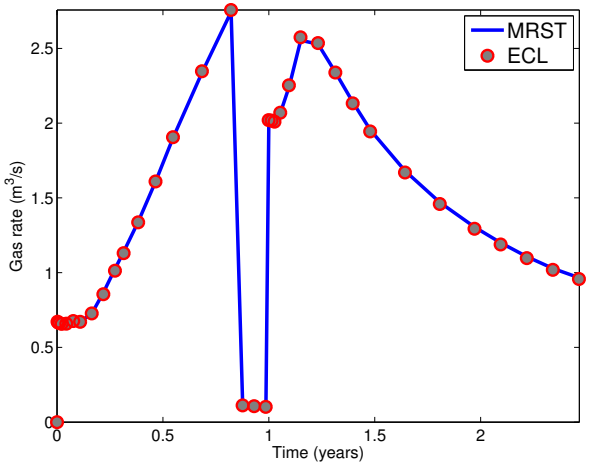
Here, you can plot bottom-hole pressures, reservoir and surface rates, oil and water cut, gas-oil ratio, etc. Plots are versus time or time step, and can be instantaneous or cummulative.

J. E. Killough (1995). Ninth SPE comparative solution project: A reexamination of black-oil simulation, doi: 10.2118/29110-MS

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

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:

```
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);
```

J. E. Killough (1995). Ninth SPE comparative solution project: A reexamination of black-oil simulation, doi: 10.2118/29110-MS

Computer exercises

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