Reservoir Modelling Using MATLAB - The MATLAB Reservoir Simulation Toolbox (MRST)

Knut-Andreas Lie

SINTEF Digital, Oslo, Norway

MATLAB Energy Conference, 17–18 November 2020
MATLAB Reservoir Simulation Toolbox (MRST)
Transforming research on reservoir modelling

Unique prototyping platform:
- Standard data formats
- Data structures/library routines
- Fully unstructured grids
- Rapid prototyping:
  - Differentiation operators
  - Automatic differentiation
  - Object-oriented framework
  - State functions
- Industry-standard simulation

% Three-phase template model
fluid = initSimpleADIFluid('mu', [1, 5, 0]*centi*pascal,
   'rho', [1000, 700, 0]*kilogram/meter^-3, 'n',
   ...
   % Constant oil compressibility
fluid.b0 = @(p, varargin) exp((p/barsa)*varargin(5));

% Construct reservoir model
model = TwoPhaseOilWaterModel(G, ...
    % Define initial state
region = getInitializationRegion('datum_depth', depth,
    ...
    state0 = initStateBlackOilFluid(model, region,
    ...
    % Define schedule
schedule = simpleSchedule(timesteps, 'W', 1);
MATLAB Reservoir Simulation Toolbox (MRST)

Transforming research on reservoir modelling

Large international user base:
- downloads from the whole world
- 123 master theses
- 56 PhD theses
- 226 journal papers (not by us)
- 144 proceedings papers

Numbers are from Google Scholar notifications

Used both by academia and industry

Google Analytics: access pattern for www.mrst.no
Period: 1 July 2018 to 31 December 2019
Different development process:

- Use abstractions to express your ideas in a form close to the underlying mathematics
- Build your program using an interactive environment:
  - try out each operation and build program as you go
- Dynamic type checking lets you prototype while you test an existing program:
  - 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

- MATLAB is fairly efficient using vectorization, logical indexing, external iterative solvers, etc.
- Avoids build process, linking libraries, cross-platform problems
- Built-in mathematical abstractions, numerics, data analysis, visualization, debugging/profiling,
- Use scripting language as a wrapper when you develop solvers in compiled languages
Modular design:

- **small core** with mature and well-tested functionality used in many programs or modules
- **semi-independent modules** that extend/override core functionality
- in-source documentation like in MATLAB
- all modules must have code examples and/or tutorials
- new development: project → module

This simplifies how we distinguish public and in-house or client-specific functionality
Latest release: 51 modules

- Grid generation and coarsening
- ECLIPSE input and output
- Upscaling / multiscale solvers
- Consistent discretizations
- Black-oil, EOR, compositional
- Fractures: DFM, EDFM, DPDP
- Geomechanics, geochemistry, geothermal
- Unsaturated media (Richards eq.)
- Multisegment wells (general network)
- CO2 storage laboratory
- Adjoints, optimization, ensembles
- Pre/postprocessing/visualization
- Flow diagnostics
- ...

3000 files, 213 000 code lines
User resources (getting help)

- website
- user forum
- textbook
- manpages
- tutorial codes
- online tutorials
Fully unstructured grids

A wide variety of grid formats:
- Cartesian and rectilinear
- Corner-point
- Tetrahedral, prismatic, PEBI
- General polyhedral/polytopal
- Hybrid, cut-cell, or depogrids
- Local refinements . . .

MRST grids are chosen to always be fully unstructured
→ can implement algorithms without knowing the specifics of the grid

Also: coarse grids made as static or dynamic partitions of fine grid
Incompressible flow solvers

\[ \nabla \cdot (p + \rho \vec{g}) = 0 \]

% Define the model
gravity reset on
G = cartGrid([2, 2, 30], [1, 1, 30]);
G = computeGeometry(G);
rock.perm = repmat(0.1 * darcy(), [G.cells.num, 1]);
fluid = initSingleFluid();
bc = pside([], G, 'TOP', 1:G.cartDims(1), ...
    1:G.cartDims(2), 100.*barsa());

% Assemble and solve the linear system
S = computeMimeticIP(G, rock);
sol = solveIncompFlow(initResSol(G, 0.0), ...
    initWellSol([], 0.0), ...
    G, S, fluid, 'bc', bc);

% Plot the face pressures
newplot;
plotFaces(G, 1:G.faces.num, sol.facePressure./barsa);
set(gca, 'ZDir', 'reverse'), title('Pressure [bar]')
view(3), colorbar

Oldest part of MRST:

- Procedural programming
- Structs for reservoir state, rock parameters, wells, b.c., and source term
- Fluid behavior: struct with function pointers

Advantages:

- hide specific details of geomodel and fluid model
- vectorization: efficient/compact code
- unified access to key parameters
Rapid prototyping: discrete differentiation operators

Grid structure in MRST

<table>
<thead>
<tr>
<th>c</th>
<th>F(c)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Map: cell → faces

<table>
<thead>
<tr>
<th>f</th>
<th>C₁</th>
<th>C₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

Map: face → cells

Idealized models

Industry models
For finite volumes, discrete $\nabla$ operator maps from cell pair $C_1(f), C_2(f)$ to face $f$:

$$\nabla (p)[f] = p[C_2(f)] - p[C_1(f)],$$

where $p[c]$ is a scalar quantity associated with cell $c$. Discrete $\text{div}$ maps from faces to cells.

Both are linear operators and can be represented as sparse matrix multiplications.
Incompressible flow:
\[ \nabla \cdot (K \nabla p) + q = 0 \]
Compressible flow:
\[ \frac{\partial (\phi \rho)}{\partial t} + \nabla \cdot (\rho K \nabla p) + q = 0 \]

Discrete in MATLAB
Incompressible flow:
\[ \text{eq} = \text{div}(T .* \text{grad}(p)) + q; \]
Compressible flow:
\[ \text{eq} = (pv(p) .* \text{rho}(p) - pv(p0) .* \text{rho}(p0)) / \text{dt} \ldots \\
+ \text{div}((\text{avg}(	ext{rho}(p)) .* T .* \text{grad}(p)) + q; \]
Discretization of flow models leads to large systems of nonlinear equations. Can be linearized and solved with Newton’s method

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

Coding necessary Jacobians is time-consuming and error prone
Automatic differentiation

General idea:
- Any code consists of a limited set of arithmetic operations and elementary functions
- Introduce an extended pair, \( \langle x, 1 \rangle \), i.e., the value \( x \) and its derivative 1
- Use chain rule and elementary derivative rules to mechanically accumulate derivatives at specific values of \( x \)
  - Elementary: \( v = \sin(x) \) \( \rightarrow \) \( \langle v \rangle = \langle \sin x, \cos x \rangle \)
  - Arithmetic: \( v = fg \) \( \rightarrow \) \( \langle v \rangle = \langle fg, fg_x + f_x g \rangle \)
  - Chain rule: \( v = \exp(f(x)) \) \( \rightarrow \) \( \langle v \rangle = \langle \exp(f(x)), \exp(f(x))f'(x) \rangle \)
- Use operator overloading to avoid messing up code

\[
[x,y] = \text{initVariablesADI}(1,2);
\]
\[
z = 3*\exp(-x*y)
\]
Example: incompressible single-phase flow

% Make grid
G = twister(cartGrid([8 8]));
G = computeGeometry(G);

% Set source terms (flow SW -> NE)
q = zeros(G.cells.num,1);
q([1 end]) = [1 -1];

% Unit isotropic permeability
K = ones(G.cells.num,4); K(:,[2 3]) = 0;

% Make grid using external grid generator
pv = [-1 -1; 0 -.5; 1 -1; 1 1; 0 .5; -1 1; -1 -1];
fh = @(p,x) 0.025 + 0.375*sum(p.^2,2);
[p,t] = distmesh2d(@(p)@dpoly, fh, 0.025, [-1 -1; 1 1], pv, pv);
G = computeGeometry(pshi(triangleGrid(p, t)));

% Set source terms (flow SW -> NE)
q = zeros(G.cells.num,1);
v = sum(G.cells.centroids,2);
[i1 i2] = min(v); [i1 i2] = max(v);
q([i1 i2]) = [1 -1];

S = setupOperatorsTPFA(G, rock); % Define Div, Grad, etc
p = initVariablesADI(zeros(G.cells.num,1)); % Initialize p as ADI variable
eq = S.Div(S.T * S.Grad(p)) + q; % Residual equation: F = Ap + q

eq(1) = eq(1) + p(1); % Fixate pressure
p = -eq.jac{1} \eq.val; % Solve system A
Example: incompressible single-phase flow

% Make grid
G = twister(cartGrid([8 8]));
G = computeGeometry(G);

% Set source terms (flow SW -> NE)
q = zeros(G.cells.num,1);
q([1 end]) = [1 -1];

% Unit isotropic permeability
K = ones(G.cells.num,4); K(:,[2 3]) = 0;

% Make grid using external grid generator
pv = [-1 -1; 0 -.5; 1 -1; 1 1; 0 .5; -1 1; -1 -1];
fh = @(p,x) 0.025 + 0.375*sum(p.^2,2);
[p,t] = distmesh2d(@dpoly, fh, 0.025, [-1 -1; 1 1], pv, pv);
G = computeGeometry(petri(triangleGrid(p, t)));

% Set source terms (flow SW -> NE)
q = zeros(G.cells.num,1);
v = sum(G.cells.centroids,2);
[~,i1] = min(v);
[~,i2] = max(v);
q([i1 i2]) = [1 -1];

S = setupOperatorsTPFA(G,rock); % Define Div, Grad, etc
p = initVariablesADI(zeros(G.cells.num,1)); % Initialize p as AD variable
eq = S.Div(S.T.*S.Grad(p)) + q; % Residual equation: F = Ap + q
eq(1) = eq(1) + p(1); % Fixate pressure
p = -eq.jac{1}

% Solve system A
Example: compressible two-phase flow

$[p, sW] = \text{initVariablesADI}(p0, sW0);$  
$[pIx, sIx] = \text{deal}(1:nc, (nc+1):(2*nc));$  
$[tol, maxits] = \text{deal}(1e-5, 15);$  
t = 0;$  
\text{while } t < \text{totTime,}$  
\quad t = t + dt;$  
\quad resNorm = 1e99; nit=0;$  
\quad [p0, sW0] = \text{deal(value(p)}, \text{value}(p));$  
\text{while } (\text{resNorm} > \text{tol}) \&\& (\text{nit} < \text{maxits}),$  
\quad % one Newton iteration$  
\text{end}$  
\text{if } nit > \text{maxits,}$  
\quad \text{error}(\text{'Newton solves did not converge'});$  
\text{end}$  
\text{end}$

% Evaluate equations
$[rW, r0, vol] = \text{deal(rhoW(p), rhoO(p), pv(p))};$  
\text{:}$  
\text{water} = (\text{vol.}*rW.*sW - \text{vol0.}*rW0.*sW0)/\text{dt} + \text{div(vW)};$  
\text{water(inIx) = water(inIx) - inRate.*rhoWS};$  
\text{:}$

\text{eqs} = \{\text{oil, water}\};$  
\text{eq} = \text{cat(eqs{:});}$  
\text{res} = \text{eq.val;}$  
\text{upd} = -(\text{eq.jac}{1} \setminus \text{res});$  
\text{% Newton update}$

% Update variables
$p.\text{val} = p.\text{val} + \text{upd(pIx)};$  
$sW.\text{val} = sW.\text{val} + \text{upd(sIx)};$  
$sW.\text{val} = \text{max( min(sW.\text{val}, 1), 0);}$  
\text{resNorm = norm(res);}  
\text{nit} = \text{nit + 1};$
The AD-OO simulator framework

**Initial state**

Solves *simulation schedule* comprised of time steps and drive mechanisms (wells/bc)

simulateScheduleAD

**Physical model**

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

TwoPhaseOilWaterModel, ThreePhaseBlackOilModel

**Result handler**

Stores and retrieves simulation data from memory/disk in a transparent and efficient manner.

**Visualization**

Visualize well curves, reservoir properties, etc

plotCellData, plotToolbar, plotWellSols, ...

**Schedule**

Steps

Time step and control numbers

\{ (\Delta T_i, C_i) \}, \ldots, \{ (\Delta T_n, C_n) \}

**State**

Initial state

State \( (T_i) \), \( \Delta T_i \), Controls \( (C_i) \)

State \( (T_i + \Delta T_i) \)

Nonlinear solver

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

Time step selector

Determines optimal time steps

SimpleTimeStepSelector, IterationCountSelector, StateChangeTimeStepSelector, ...

Time step and control numbers

\{ (\Delta T_i, C_i) \}, \ldots, \{ (\Delta T_n, C_n) \}

**Primary vars**

[Res, Jac], info Assemble: \( Ax = b \)

Update variables:

\[ p \leftarrow p + \delta p, s \leftarrow s + \delta s, \ldots \]

**Well solutions**

Well data:

\( q_W, q_Q, q_G, bhp, \ldots \)

**Well model**

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

**Linear solver**

Solves linearized problem and returns increments

BackslashSolverAD, AGMGSolverAD, CPRSolverAD, MultiscaleSolverAD, ...

Assemble: \( Ax = b \)

JacOrians, residual equations and meta-information about their types

**Physical model**

Well model

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

[Res, Jac], info

**Linearized problem**

Well curves

3D visualization

Write to storage

Type color legend

Class

Struct

Function(s)

Input

Contains object

Optional output
Capabilities as in commercial simulators

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

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

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

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

Wells
- Physical variables inside the wellbore
  - \( q_w, q_d, q_g, q_o, p_{bh} \)

Example:
- Field production compared with OPM Flow for the Norne field
It would be convenient to have:

- **Dependency management**: keep track of dependency graph, ensure all input quantities have been evaluated before evaluating a function.

- **Generic interfaces**: avoid defining functional dependencies explicitly, e.g., $G(S)$, and $G(p, S)$.

- Lazy evaluation with caching

- Enable spatial dependence in parameters while preserving vectorization potential

- Implementation independent of the choice of primary variables
It would be convenient to have:

- **Dependency management**: keep track of dependency graph, ensure all input quantities have been evaluated before evaluating a function

- **Generic interfaces**: avoid defining functional dependencies explicitly, e.g., $G(S)$, and $G(p, S)$.

- Lazy evaluation with caching

- Enable spatial dependence in parameters while preserving vectorization potential

- Implementation independent of the choice of primary variables

**State function**: any function that is uniquely determined by the contents of the state struct alone

Implemented as class objects, gathered in functional groups

\[ G(x, y, a, b) = xy + ab \]
Example: State-function diagram for a simple black-oil model. Each entity is a state function that is easy to replace.

Idea: apply this concept to
- flow property evaluation
- PVT calculations
- accumulation, flux, and source terms
- spatial/temporal discretization

Simulator $\rightarrow$ differentiable graph

Further granularity
- Immiscible components
- Black-oil type components
- Compositional components
- Concentration components

Combined at runtime to form compact models with only necessary unknowns
What about computational performance?

Total time of a program consists of several parts:

programming + debugging
    + documenting + testing + executing

MRST is designed to prioritize the first four over the last

Does this mean that MRST is slow and not scalable?

Potential concerns:
MATLAB is interpreted
cure: JIT, vectorization, logical indexing, pre-allocation, highly-efficient libraries
Redundant computations
cure: state functions = dependency graph + computational cache
Computational overhead
cure: new auto diff backends
Scalability/performance
cure: external high-end iterative solvers
What about computational performance?

Total time of a program consists of several parts:

- programming + debugging
- + documenting + testing + executing

MRST is designed to prioritize the first four over the last

Does this mean that MRST is slow and not scalable?

No, I would say its is surprisingly efficient

Potential concerns:

- MATLAB is interpreted
  cure: JIT, vectorization, logical indexing, pre-allocation, highly-efficient libraries

- Redundant computations
  cure: state functions = dependency graph + computational cache

- Computational overhead
  cure: new auto diff backends

- Scalability/performance
  cure: external high-end iterative solvers
New backends for automatic differentiation

New AD backends: storage optimized wrt access pattern, MEX-accelerated operations
Interface to external linear algebra packages implemented as classes in AD-OO framework

**Example:** compressible three-phase, black-oil problem

<table>
<thead>
<tr>
<th>Solver</th>
<th>Req.</th>
<th>8,000 cells</th>
<th>125,000 cells</th>
<th>421,875 cells</th>
<th>1,000,000 cells</th>
</tr>
</thead>
<tbody>
<tr>
<td>LU</td>
<td>–</td>
<td>2.49 s</td>
<td>576.58 s</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>CPR*</td>
<td>–</td>
<td>0.90 s</td>
<td>137.30 s</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>CPR*</td>
<td>AGMG</td>
<td>0.18 s</td>
<td>3.60 s</td>
<td>13.78 s</td>
<td>43.39 s</td>
</tr>
<tr>
<td>CPR*</td>
<td>AMGCL</td>
<td>0.21 s</td>
<td>3.44 s</td>
<td>16.20 s</td>
<td>51.35 s</td>
</tr>
<tr>
<td>CPR</td>
<td>AMGCL</td>
<td>0.07 s</td>
<td>0.43 s</td>
<td>3.38 s</td>
<td>10.20 s</td>
</tr>
<tr>
<td>CPR</td>
<td>AMGCL$\dagger$</td>
<td>0.05 s</td>
<td>0.86 s</td>
<td>1.97 s</td>
<td>5.60 s</td>
</tr>
<tr>
<td>CPR</td>
<td>AMGCL$\ddagger$</td>
<td>0.05 s</td>
<td>0.38 s</td>
<td>1.33 s</td>
<td>3.82 s</td>
</tr>
</tbody>
</table>

* – in MATLAB, $\dagger$ – block AMGCL (block ILU + AMG/CPR), $\ddagger$ – block AMGCL with tweaks

Performance is approaching commercial and compiled academic codes
New book: Advanced modelling with MRST

Berge et al.: Constrained Voronoi grids

Al Kobaisi & Zhang: nonlinear FVM

Lie & Møyner: multiscale methods

Wong et al.: embedded discrete fractures

Olorode et al.; fractured unconventional

March et al.: unified framework, fractures

Varela et al.: unsaturated poroelasticity

Collignon et al.: geothermal systems

Klemetsdal & Lie: discontinuous Galerkin

Møyner: state functions, AD backends

Sun et al.: chemical EOR

Møyner: compositional

Andersen: coupled flow & geomechanics
Thanks to all my co-developers at SINTEF (Olav Møyner, in particular), our master and PhD students, and our national and international collaborators.

Thanks also to all MRST users who have asked interesting questions that have helped us shape the software

Funding:
- Research Council of Norway
- SINTEF
- Equinor: gold open access for the MRST textbook
- Chevron, Ecopetrol, Eni, Equinor, ExxonMobil, Shell, SLB, Total, Wintershall DEA, . . .