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

Knut-Andreas Lie

SINTEF Digital, Oslo, Norway

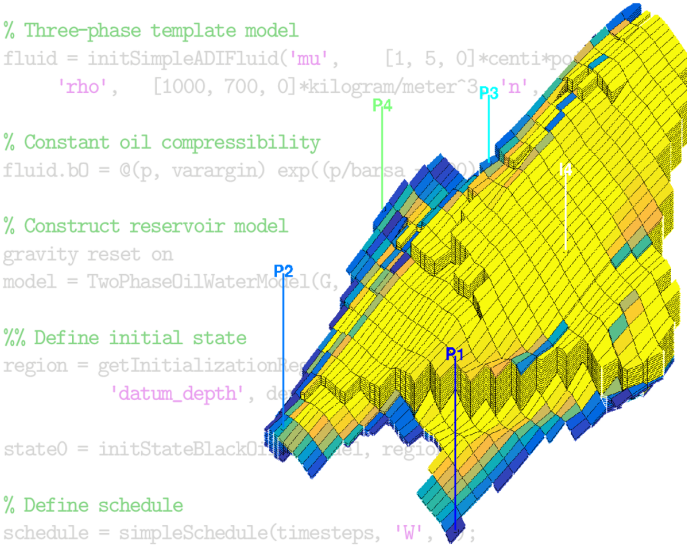
MATLAB Energy Conference, 17–18 November 2020

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



http://www.mrst.no

MATLAB Reservoir Simulation Toolbox (MRST)

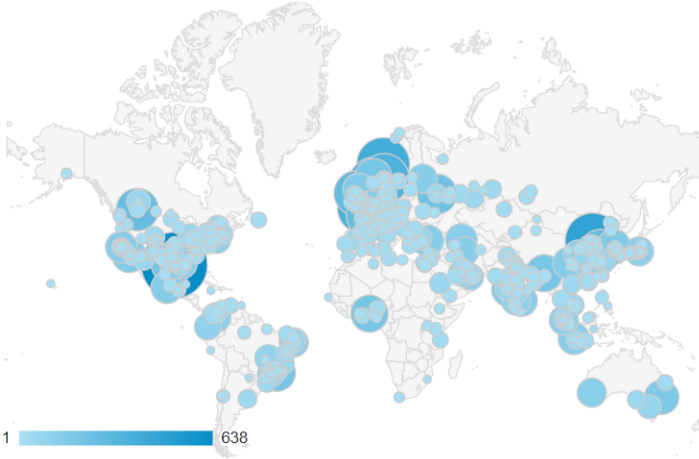
Transforming research on reservoir modelling

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

Reservoir simulation in MATLAB...?

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

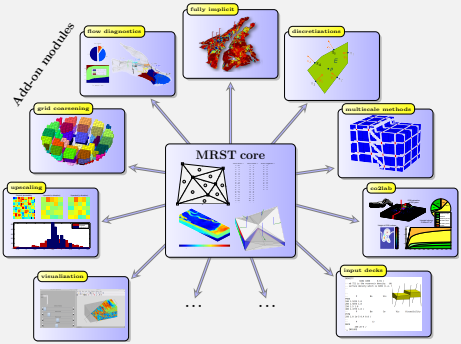
- MATLAB is fairly efficient using vectorization, logical indexing, external iterative solvers, etc.
- Avoids build process, linking libraries, cross-platform problems
- Builtin mathematical abstractions, numerics, data analysis, visualization, debugging/profiling,
- Use scripting language as a wrapper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

Community code: software organization

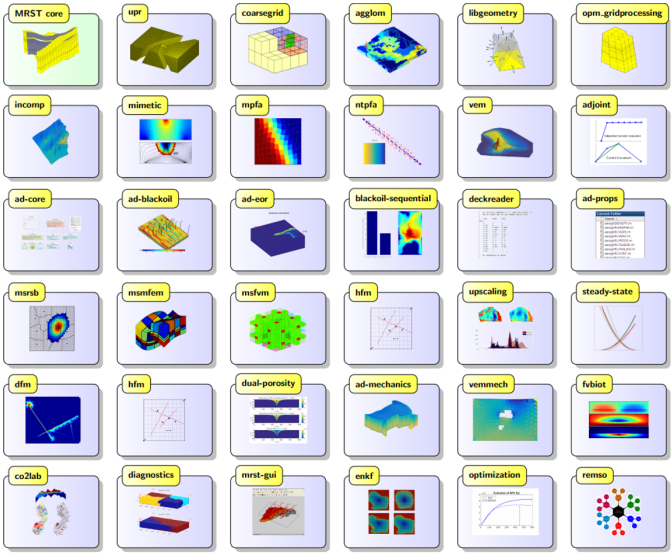
Modular design:

- **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 $\longrightarrow$ module

This simplifies how we distinguish public and in-house or client-specific functionality



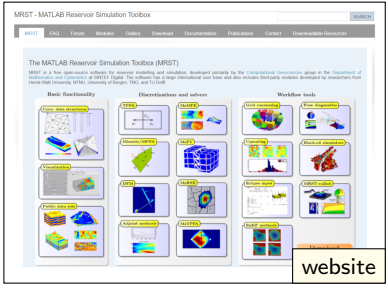
Latest release: 51 modules



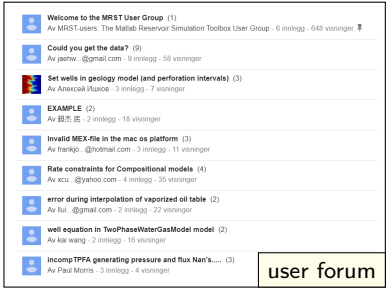
- Grid generation and coarsening
- ECLIPSE input and output
- Upscaling / multiscale solvers
- Consistent discretizations
- Black-oil, EOR, compositional
- Fractures: DFM, EDFM, DPDP
- Geomechanics, geochemistry, geothermal
- Unsaturated media (Richards eq.)
- Multisegment wells (general network)
- CO2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

3000 files, 213 000 code lines

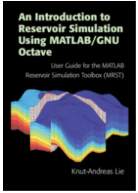
User resources (getting help)



website



user forum



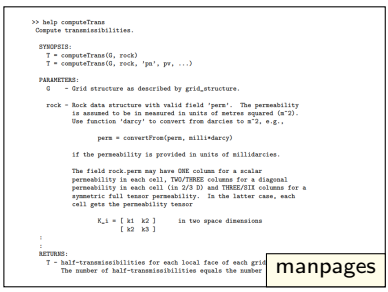
Access Open access

Knut-Andreas Lie, *SINTEF, Norway*

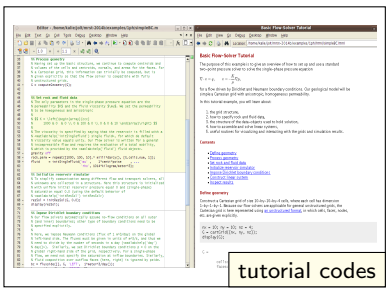
Publisher: Cambridge University Press
Online publication date: July 2019
Print publication year: 2019
Online ISBN: 9781108591416

DOI: https://doi.org/10.1017/9781108591416

textbook



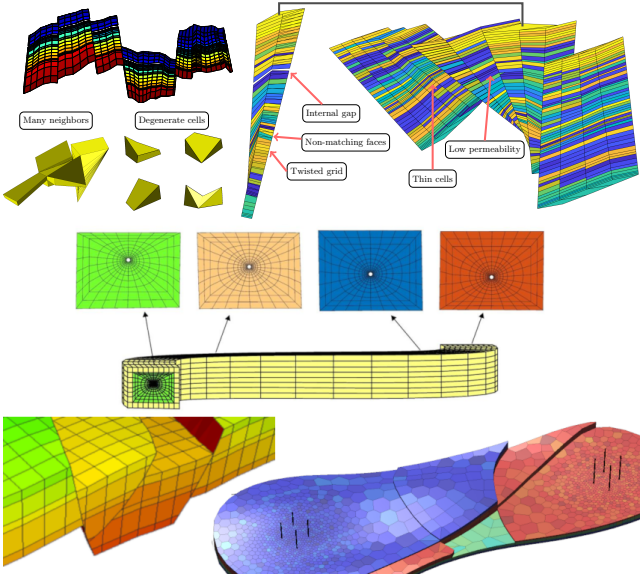
manpages



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 . . .

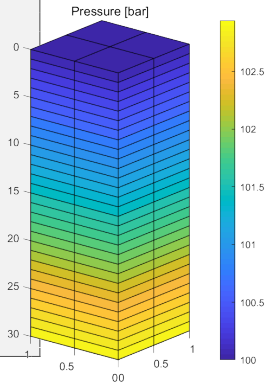
MRST grids are chosen to always be **fully unstructured**
$\longrightarrow$ can implement algorithms without knowing the specifics of the grid

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

Incompressible flow solvers

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

```
%% 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:

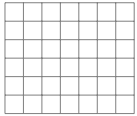
- Procedural programming
- Structs for reservoir state, rock parameters, wells, b.c., and source term
- Fluid behavior: struct with function pointers

Advantages:

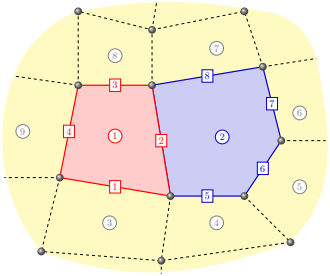
- **hide specific details** of geomodel and fluid model
- **vectorization**: efficient/compact code
- **unified access** to key parameters

Rapid prototyping: discrete differentiation operators

Idealized models



Grid structure in MRST



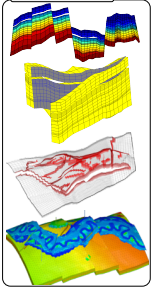
c	F(c)
1	1
1	2
1	3
1	4
2	5
2	6
2	7
2	8
2	2
3	1
⋮	⋮
⋮	⋮

Map: cell → faces

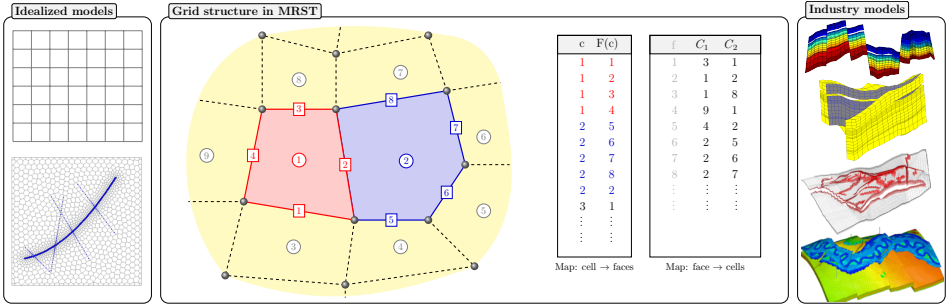
f	C_1	C_2
1	3	1
2	1	2
3	1	8
4	9	1
5	4	2
6	2	5
7	2	6
8	2	7
⋮	⋮	⋮
⋮	⋮	⋮

Map: face → cells

Industry models



Rapid prototyping: discrete differentiation operators



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

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
⋮	⋮	⋮

For finite volumes, 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 `div` maps from faces to cells

Both are linear operators and can be represented as sparse matrix multiplications

Close correspondence with mathematics

Continuous

Incompressible flow:

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

Compressible flow:

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

Discrete in MATLAB

Incompressible flow:

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

Compressible flow:

```
eq = (pv(p).*rho(p)-pv(p0).*rho(p0))/dt ...
     + div(avg(rho(p)).*T.*grad(p))+q;
```

Close correspondence with mathematics

Continuous

Incompressible flow:

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

Compressible flow:

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

Discrete in MATLAB

Incompressible flow:

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

Compressible flow:

```
eq = (pv(p).*rho(p)-pv(p0).*rho(p0))/dt ...
     + div(avg(rho(p)).*T.*grad(p))+q;
```

Discretization of flow models leads to large systems of nonlinear equations. Can be linearized and solved with Newton's method

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

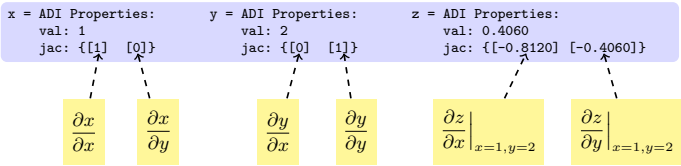
Coding necessary Jacobians is time-consuming and error prone

Automatic differentiation

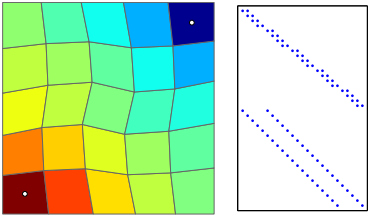
General idea:

- 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: $v = \sin(x) \longrightarrow \langle v\rangle = \langle \sin x, \cos x\rangle$
  - Arithmetic: $v = fg \longrightarrow \langle v\rangle = \langle fg, fg_x + f_x g\rangle$
  - Chain rule: $v = \exp(f(x)) \longrightarrow \langle v\rangle = \langle \exp(f(x)), \exp(f(x))f'(x)\rangle$
- Use operator overloading to avoid messing up code

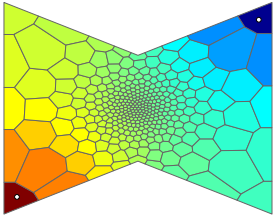
```
[x,y] = initVariablesADI(1,2);
z = 3*exp(-x*y)
```

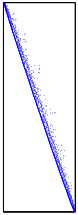


Example: incompressible single-phase flow



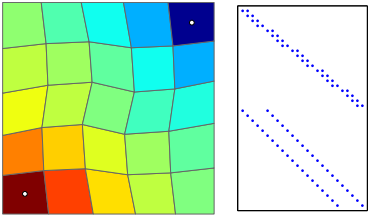

```
% Make grid
G = twister(cartGrid([8 8]));
G = computeGeometry(G);

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

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

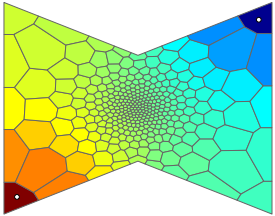


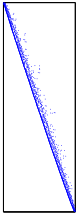


```
% Make grid using external grid generator
pv = [-1 -1; 0 -.5; 1 -1; 1 1; 0 .5; -1 1; -1 -1];
fh = @(p,x) 0.025 + 0.375*sum(p.^2,2);
[p,t] = distmesh2d(@dpoly, fh, 0.025, [-1 -1; 1 1], pv, pv);
G = computeGeometry(pebi(triangleGrid(p, t)));

% 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];
```

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 insotropic permeability
K = ones(G.cells.num,4); K(:,[2 3]) = 0;
```

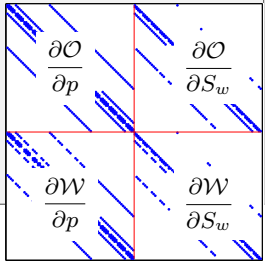
```
% Make grid using external grid generator
pv = [-1 -1; 0 -.5; 1 -1; 1 1; 0 .5; -1 1; -1 -1];
fh = @(p,x) 0.025 + 0.375*sum(p.^2,2);
[p,t] = distmesh2d(@dpoly, fh, 0.025, [-1 -1; 1 1], pv, pv);
G = computeGeometry(pebi(triangleGrid(p, t)));

% 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}\eq.val;                             % Solve system A
```

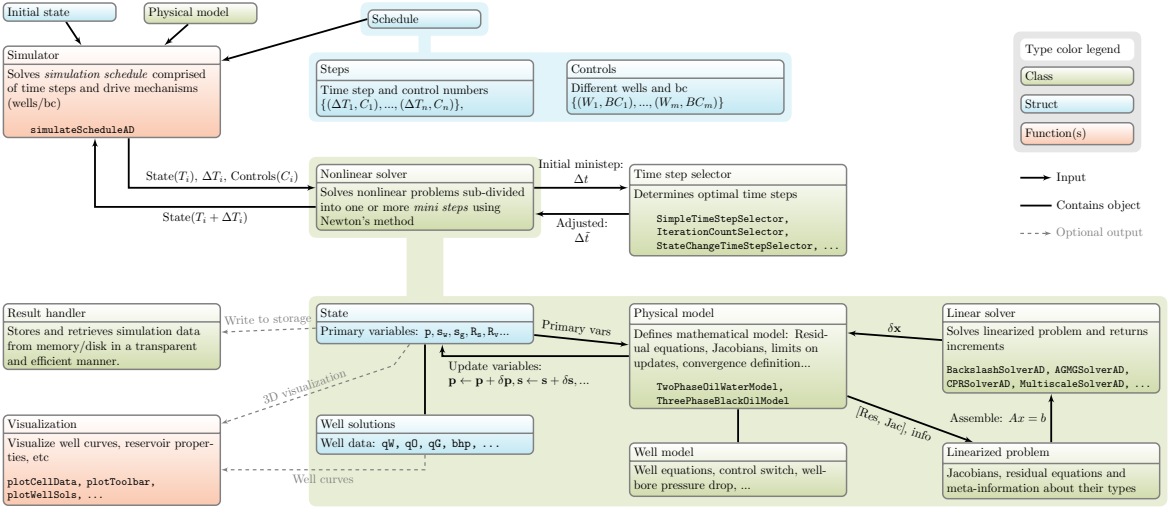
Example: compressible two-phase flow


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

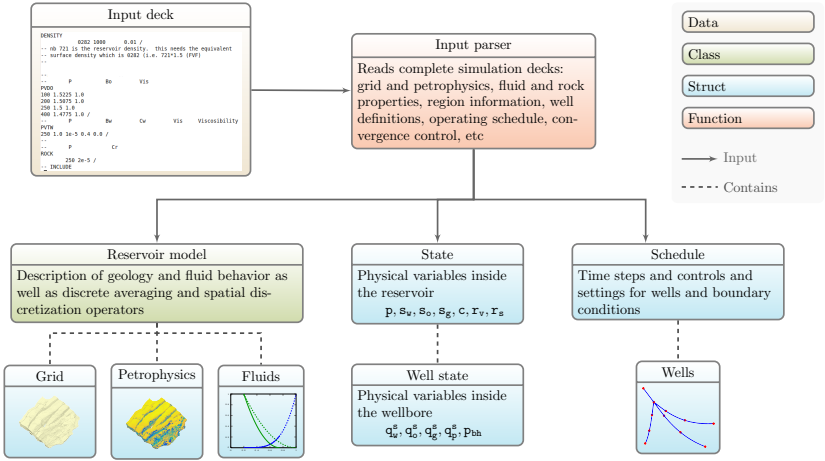
```
% Evaluate equations
[rW, rO, vol] = deal(rhoW(p), rhoO(p), pv(p)));
:
water = (vol.*rW.*sW - vol0.*rW0.*sW0)./dt + div(vW);
water(inIx) = water(inIx) - inRate.*rhoWS;
:
eqs = {oil, water};          % concatenate equations
eq  = cat(eqs{:});           % assemble
res = eq.val;                % residual
upd = -(eq.jac{1} \ res);    % Newton update

% Update variables
p.val   = p.val  + upd(pIx);
sW.val  = sW.val + upd(sIx);
sW.val  = max( min(sW.val, 1), 0);
resNorm = norm(res);
nit     = nit + 1;
```



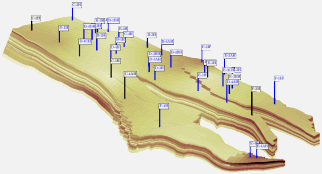
The AD-OO simulator framework

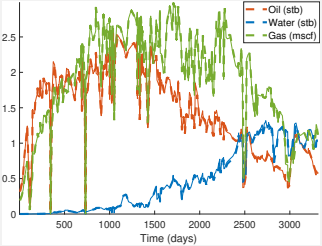


Capabilities as in commercial simulators



Example:





Field production compared with OPM Flow for the Norne field

State functions: modularity and computational cache

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

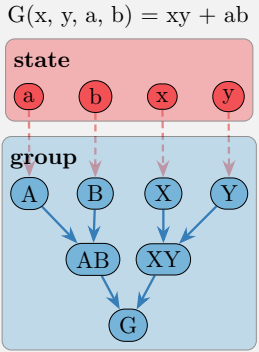
State functions: modularity and computational cache

It would be convenient to have:

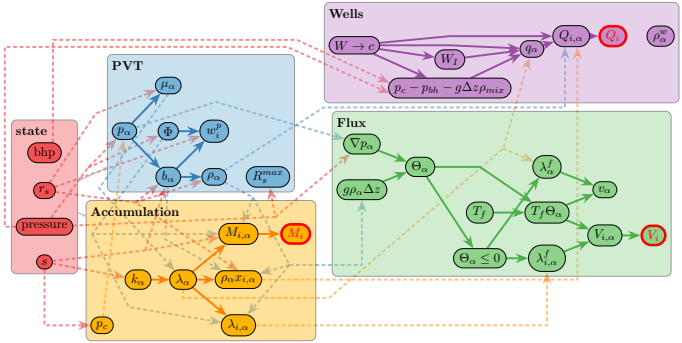
- **Dependency management**: keep track of dependency graph, ensure all input quantities have been evaluated before evaluating a function
- **Generic interfaces**: avoid defining functional dependencies explicitly, e.g., $G(S)$, and $G(p, S)$.
- Lazy evaluation with caching
- Enable spatial dependence 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



Simulator: differentiable graph



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 $\longrightarrow$ 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:

$$\text{programming} + \text{debugging} + \text{documenting} + \text{testing} + \text{executing}$$

MRST is designed to prioritize the first four over the last

Does this mean that MRST is slow and not scalable?

What about computational performance?

Total time of a program consists of several parts:

$$\text{programming} + \text{debugging} + \text{documenting} + \text{testing} + \text{executing}$$

MRST is designed to prioritize the first four over the last

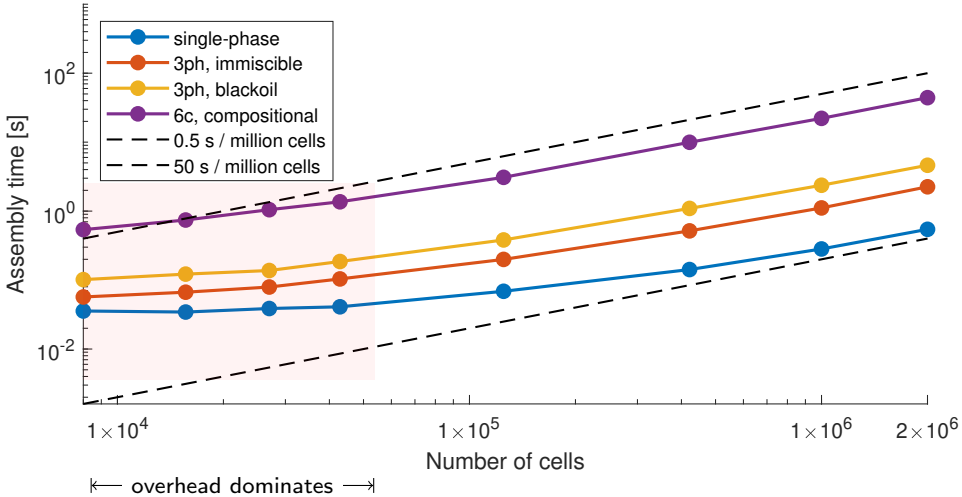
Does this mean that MRST is slow and not scalable?

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

Efficient linear solvers

Interface to external linear algebra packages implemented as classes in AD-OO framework

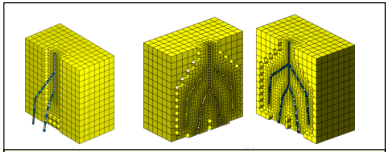
Example: compressible three-phase, black-oil problem

Solver	Req.	8,000 cells	125,000 cells	421,875 cells	1,000,000 cells
LU	–	2.49 s	576.58 s	–	–
CPR*	–	0.90 s	137.30 s	–	–
CPR*	AGMG	0.18 s	3.60 s	13.78 s	43.39 s
CPR*	AMGCL	0.21 s	3.44 s	16.20 s	51.35 s
CPR	AMGCL	0.07 s	0.43 s	3.38 s	10.20 s
CPR	AMGCL†	0.05 s	0.86 s	1.97 s	5.60 s
CPR	AMGCL‡	0.05 s	0.38 s	1.33 s	3.82 s

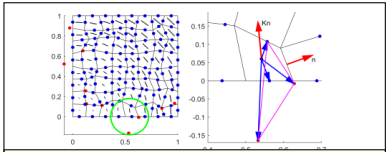
$*$ – in MATLAB, † – block AMGCL (block ILU + AMG/CPR), ‡ – block AMGCL with tweaks

Performance is approaching commercial and compiled academic codes

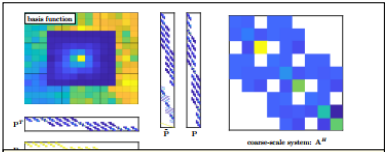
New book: Advanced modelling with MRST



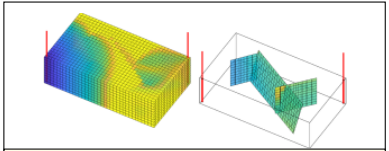
Berge et al.: Constrained Voronoi grids



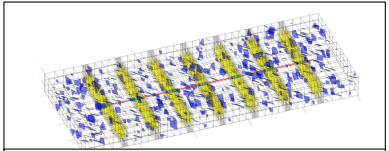
Al Kobaisi & Zhang: nonlinear FVM



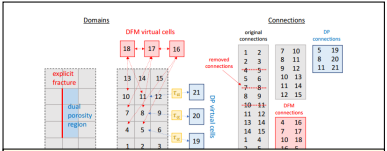
Lie & Møyner: multiscale methods



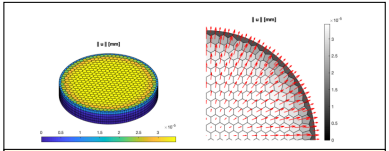
Wong et al.: embedded discrete fractures



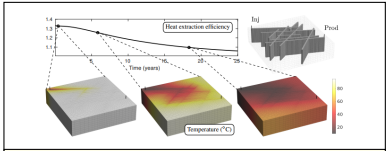
Olorode et al.; fractured unconventionals



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

Acknowledgements

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, . . .