How to use recent developments of the MATLAB Reservoir Simulation Toolbox for fast prototyping of complex fluid models

Øystein S. Klemetsdal, Olav Møyner, Halvor Møll Nilsen, Knut-Andreas Lie

SINTEF Digital, Oslo, Norway

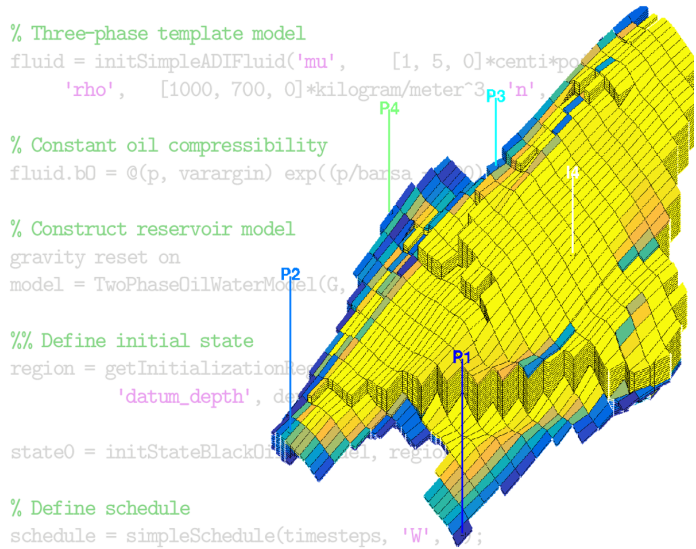
NCCS Webinar Series, October 6, 2020

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)

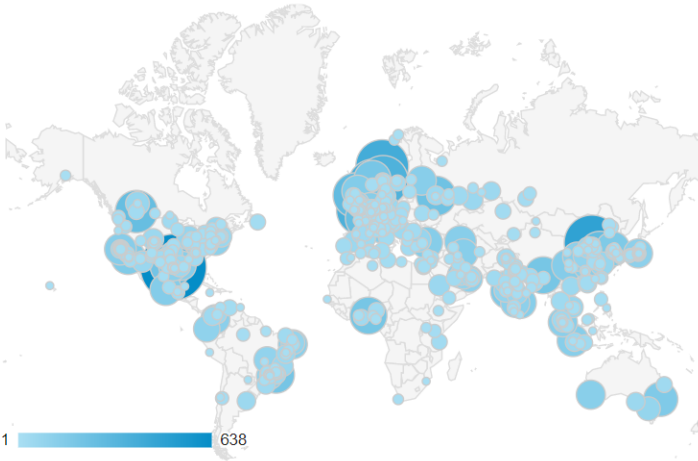
Transforming research on reservoir modelling

Large international user base:

- downloads from the whole world
- 124 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
Unique downloads: 5 516 (103 countries and 838 cities)

MATLAB Reservoir Simulation Toolbox (MRST)

Transforming research on reservoir modelling

Large international user base:

- downloads from the whole world
- 124 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

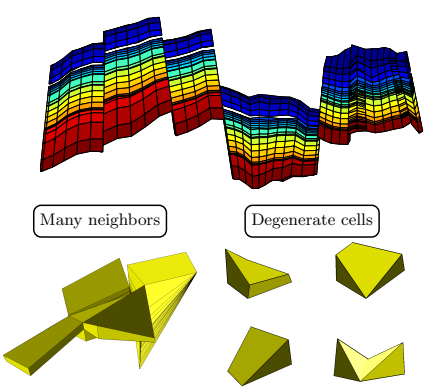
Word cloud generated from titles of journal papers

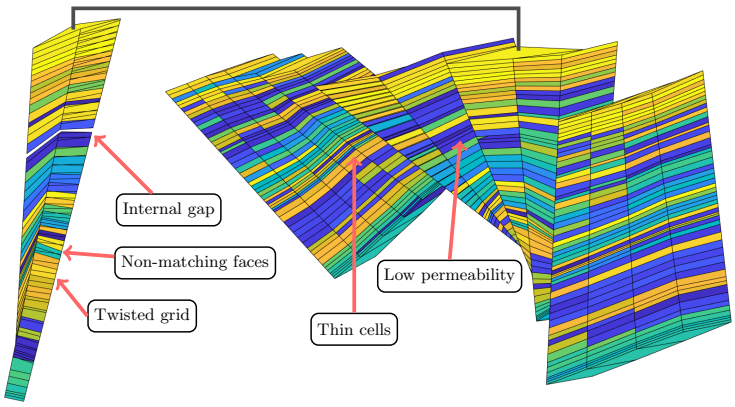
Reservoir simulation in MATLAB...?

- 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
  - extensive and flexible **debugging** capabilities
- 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
- We considered Python, but found it to be less mature (same with Julia a few years ago)

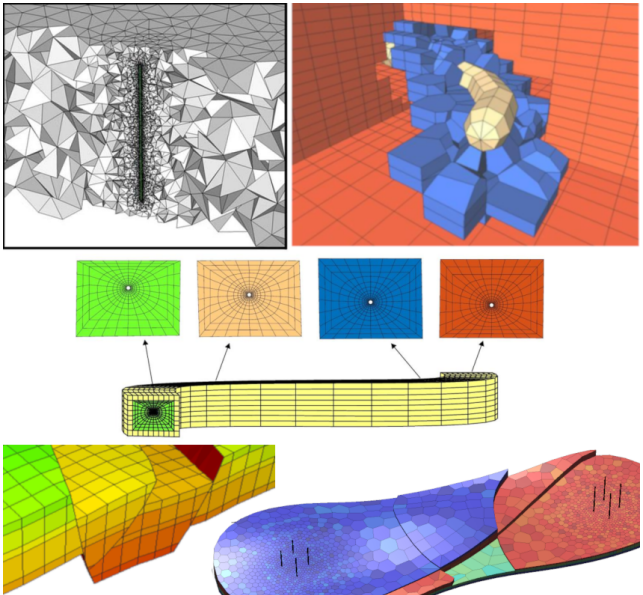
Corner-point grids (the most widespread format in industry):





Many special cases and tricks to be (re)invented

Complex, unstructured grids



A wide variety of grid formats:

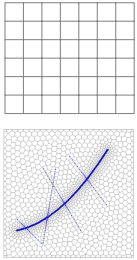
- Tetrahedral, prismatic
- PEBI
- General polyhedral/polytopal
- Hybrid, cut-cell, or depogrids
- Local refinements ...

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

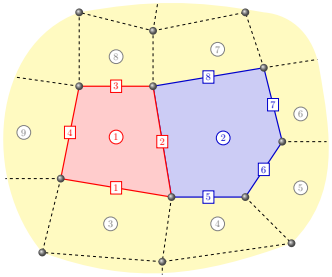
Also: coarse grids made as static or dynamic partitions of fine grid

Discrete differentiation operators

Idealized models



Grid structure in MRST



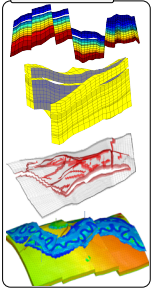
c	F(c)
1	1
1	2
1	3
1	4
2	5
2	6
2	7
2	8
2	2
3	1
⋮	⋮
⋮	⋮

Map: cell → faces

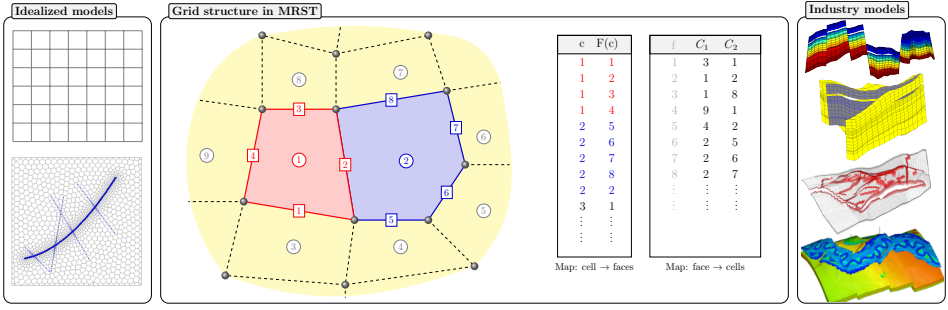
f	C_1	C_2
1	3	1
2	1	2
3	1	8
4	9	1
5	4	2
6	2	5
7	2	6
8	2	7
⋮	⋮	⋮
⋮	⋮	⋮

Map: face → cells

Industry models



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
⋮	⋮

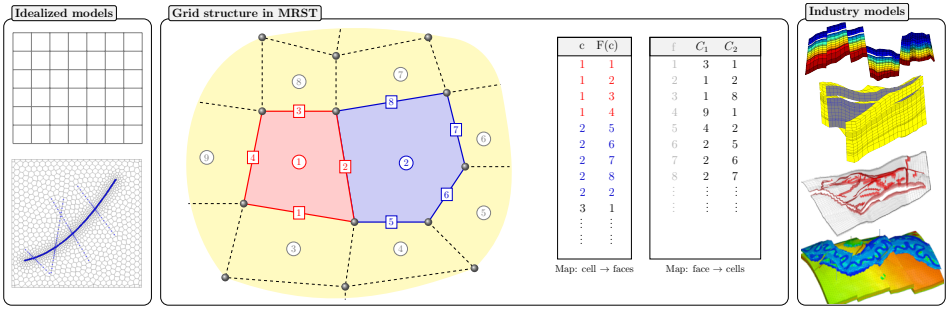
f	C_1	C_2
1	3	1
2	1	2
3	1	8
4	9	1
5	4	2
6	2	5
7	2	6
8	2	7
⋮	⋮	⋮

For finite volumes, 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



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

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

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

Coding necessary Jacobians is **time-consuming** and **error prone**

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

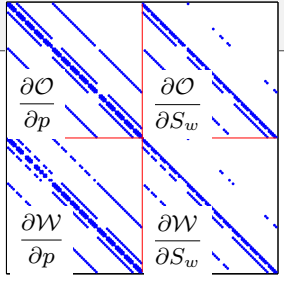
Automatic differentiation: Combine chain rule and elementary differentiation rules by means of operator overloading to efficiently evaluate all derivatives to machine precision

$\rightarrow$ Computing Jacobians amounts to writing down residual equations.

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

Residual equations are computed as for single-phase flow‡

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



‡ **Darcy's law:**
upstream operator is a combination of vector product and logical indexing

In sum: excellent approach for fast prototyping of simple simulators

In sum: excellent approach for fast prototyping of simple simulators

However, things soon start piling up

- complex rock-fluid/PVT models + hysteretic behavior
- advanced well models and simulation schedules
- time-step control, iteration control, line search, preconditioning, iterative solvers, ...

Hence, code will eventually start to become complicated

In sum: excellent approach for fast prototyping of simple simulators

However, things soon start piling up

- complex rock-fluid/PVT models + hysteretic behavior
- advanced well models and simulation schedules
- time-step control, iteration control, line search, preconditioning, iterative solvers, ...

Hence, code will eventually start to become complicated

Cure: the object-oriented AD-OO framework to separate:

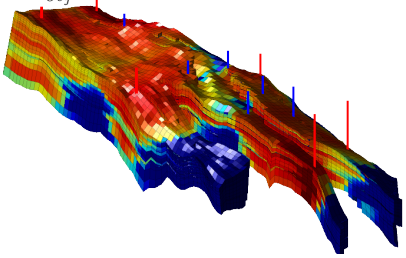
- physical models
- discretizations and discrete operators
- nonlinear solver and time-stepping
- assembly and solution of the linear system

Only expose needed details and increase reuse of developed functionality

Simulator: differentiable graph

1) Define continuous residual equations
$r_i = \frac{\partial m_i}{\partial t} + \nabla \cdot \vec{v} - q_i = 0$

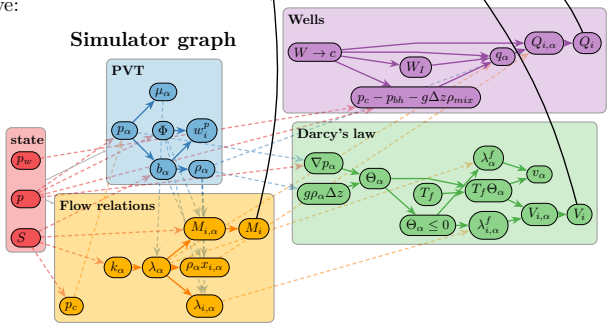
3) Differentiate discrete residual with AD and solve:
$J_{ij} = \frac{\partial R_i}{\partial v_j}, \quad x^{k+1} = x^k - J^{-1}R, \ldots$



4) Post: Make decisions, compute sensitivites, ...

2) Create *simulator graph* to discretize equations

$$R_i = \frac{1}{\Delta t}\left(M_i^{n+1} + M_i^{n} \right) + \nabla \cdot V_i - Q_i$$



Simulation on graphs	Graph of functions for multiphysics problems Easy to modify, extend and understand Smart automatic differentiation for high performance

State functions: modularity and computational cache

Similar quantities appear in multiple flow models, but can have different functional relationships

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

Similar quantities appear in multiple flow models, but can have different functional relationships

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

State functions: modularity and computational cache

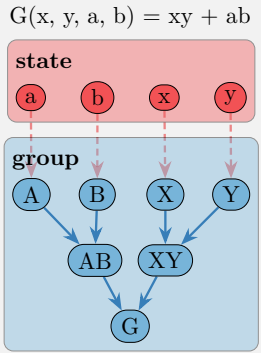
Similar quantities appear in multiple flow models, but can have different functional relationships

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



Worked example: CO_2 injection with mobility control

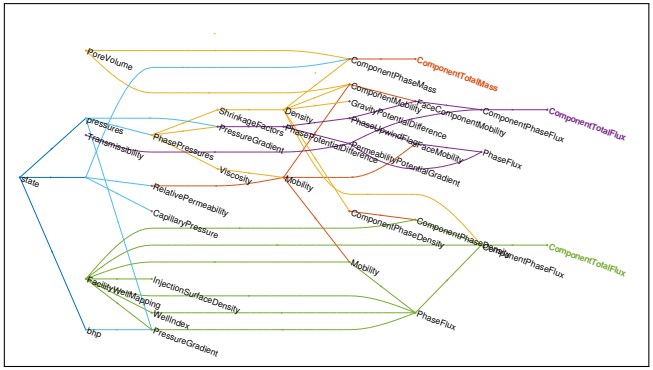
$$\frac{1}{\Delta t}(\boldsymbol{M}_\alpha^{n+1} - \boldsymbol{M}_\alpha^n) + \mathtt{div}(\boldsymbol{V}_\alpha) - \boldsymbol{Q}_\alpha = \boldsymbol{0}, \quad \alpha = w, g$$

$$\boldsymbol{V}_g = -\boldsymbol{\rho}_g \boldsymbol{\lambda}_g \boldsymbol{K} \mathtt{grad}(\boldsymbol{p}_g), \quad \boldsymbol{\lambda}_g = \frac{\boldsymbol{k}_{r,g}}{\boldsymbol{\mu}_g}$$

Worked example: CO_2 injection with mobility control

$$\frac{1}{\Delta t}(\boldsymbol{M}_\alpha^{n+1} - \boldsymbol{M}_\alpha^n) + \mathtt{div}(\boldsymbol{V}_\alpha) - \boldsymbol{Q}_\alpha = \boldsymbol{0}, \quad \alpha = w, g$$

$$\boldsymbol{V}_g = -\boldsymbol{\rho}_g \boldsymbol{\lambda}_g \boldsymbol{K} \mathtt{grad}(\boldsymbol{p}_g), \quad \boldsymbol{\lambda}_g = \frac{\boldsymbol{k}_{r,g}}{\boldsymbol{\mu}_g}$$



Worked example: CO_2 injection with mobility control

$$\frac{1}{\Delta t}(\boldsymbol{M}_\alpha^{n+1} - \boldsymbol{M}_\alpha^n) + \mathtt{div}(\boldsymbol{V}_\alpha) - \boldsymbol{Q}_\alpha = \boldsymbol{0}, \quad \alpha = w, g$$

$$\boldsymbol{V}_g = -\boldsymbol{\rho}_g \boldsymbol{\lambda}_g \boldsymbol{K} \mathtt{grad}(\boldsymbol{p}_g), \quad \boldsymbol{\lambda}_g = F(\boldsymbol{S}_g, \boldsymbol{c}) \frac{\boldsymbol{k}_{r,g}}{\boldsymbol{\mu}_g}$$

Mobility control: increase mobility of injected gas by adding surfactant
$\rightarrow$ foam formation

Modelled through mobility multiplier F[1]
$\rightarrow$ we only have to change the relevant part (`Mobility`) in the simulator graph

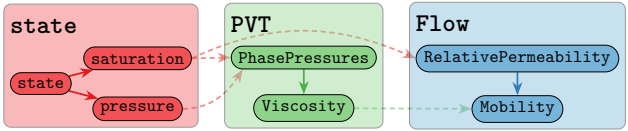
[1] Grimstad et al (2018)

Worked example: CO_2 injection with mobility control

$$\frac{1}{\Delta t}(\boldsymbol{M}_\alpha^{n+1} - \boldsymbol{M}_\alpha^n) + \mathtt{div}(\boldsymbol{V}_\alpha) - \boldsymbol{Q}_\alpha = \boldsymbol{0}, \quad \alpha = w, g$$

$$\boldsymbol{V}_g = -\boldsymbol{\rho}_g \boldsymbol{\lambda}_g \boldsymbol{K} \mathtt{grad}(\boldsymbol{p}_g), \quad \boldsymbol{\lambda}_g = F(\boldsymbol{S}_g, \boldsymbol{c}) \frac{\boldsymbol{k}_{r,g}}{\boldsymbol{\mu}_g}$$

Standard mobility




```
classdef MultipliedMobility < StateFunction
    function mob = evaluateOnDomain(prop, model, state)
        kr  = prop.getEvaluatedDependencies(state, 'RelativePermeability');
        mu  = model.getProp(state, 'Viscosity');
        mob = cellfun(@(x, y) x./y, kr, mu, 'UniformOutput', false);
    end
end
```

Mobility control: increase mobility of injected gas by adding surfactant
→ foam formation

Modelled through mobility multiplier F[1]
→ we only have to change the relevant part (`Mobility`) in the simulator graph

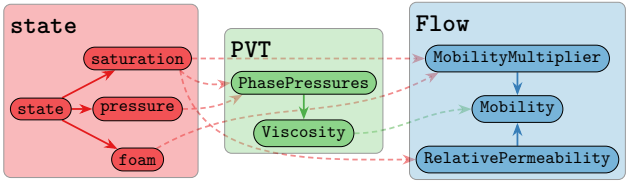
[1] Grimstad et al (2018)

Worked example: CO_2 injection with mobility control

$$\frac{1}{\Delta t}(\boldsymbol{M}_\alpha^{n+1} - \boldsymbol{M}_\alpha^n) + \mathtt{div}(\boldsymbol{V}_\alpha) - \boldsymbol{Q}_\alpha = \boldsymbol{0}, \quad \alpha = w, g$$

$$\boldsymbol{V}_g = -\boldsymbol{\rho}_g \boldsymbol{\lambda}_g \boldsymbol{K} \mathtt{grad}(\boldsymbol{p}_g), \quad \boldsymbol{\lambda}_g = F(\boldsymbol{S}_g, \boldsymbol{c}) \frac{\boldsymbol{k}_{r,g}}{\boldsymbol{\mu}_g}$$

Mobility with foam multiplier




```
classdef MultipliedMobility < Mobility
    function mob = evaluateOnDomain(prop, model, state)
        mob = evaluateOnDomain@Mobility(prop, model, state);
        F   = prop.getEvaluatedDependencies(state, 'MobilityMultiplier');
        mob{is_gas} = F.*mob{is_gas};
    end
end
```

Mobility control: increase mobility of injected gas by adding surfactant
→ foam formation

Modelled through mobility multiplier F[1]
→ we only have to change the relevant part (`Mobility`) in the simulator graph

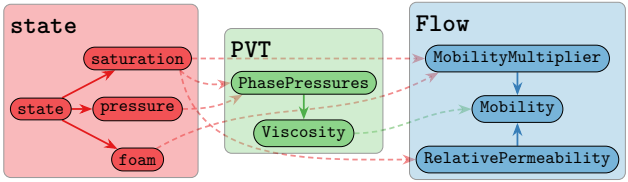
[1] Grimstad et al (2018)

Worked example: CO_2 injection with mobility control

$$\frac{1}{\Delta t}(\boldsymbol{M}_\alpha^{n+1} - \boldsymbol{M}_\alpha^n) + \mathtt{div}(\boldsymbol{V}_\alpha) - \boldsymbol{Q}_\alpha = \boldsymbol{0}, \quad \alpha = w, g$$

$$\boldsymbol{V}_g = -\boldsymbol{\rho}_g \boldsymbol{\lambda}_g \boldsymbol{K} \mathtt{grad}(\boldsymbol{p}_g), \quad \boldsymbol{\lambda}_g = F(\boldsymbol{S}_g, \boldsymbol{c}) \frac{\boldsymbol{k}_{r,g}}{\boldsymbol{\mu}_g}$$

Mobility with foam multiplier




```
classdef MultipliedMobility < Mobility
    function mob = evaluateOnDomain(prop, model, state)
        mob = evaluateOnDomain@Mobility(prop, model, state);
        F   = prop.getEvaluatedDependencies(state, 'MobilityMultiplier');
        mob{is_gas} = F.*mob{is_gas};
    end
end
```

Mobility control: increase mobility of injected gas by adding surfactant
→ foam formation

Modelled through mobility multiplier F[1]
→ we only have to change the relevant part (`Mobility`) in the simulator graph

Dependencies on other properties are defined in the implementation
→ evaluated in the correct order, and **only once** per nonlinear iteration

[1] Grimstad et al (2018)

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?

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

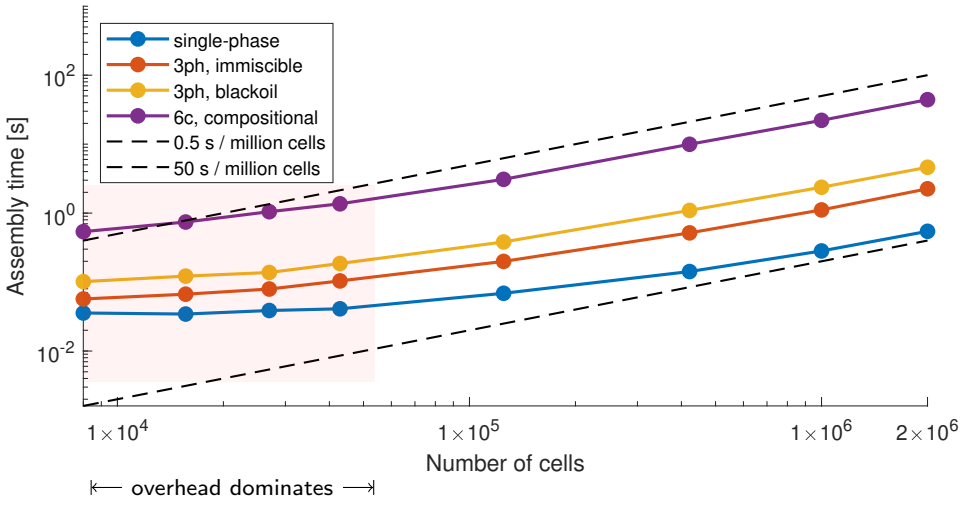
Does this mean that MRST is slow and not scalable?

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

Efficient linear solvers

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

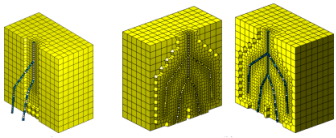
Example: compressible three-phase, black-oil problem

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

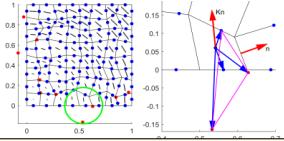
$*$ – in MATLAB, † – block AMGCL (block ILU + AMG/CPR), ‡ – block AMGCL with tweaks

Performance is approaching commercial and compiled academic codes

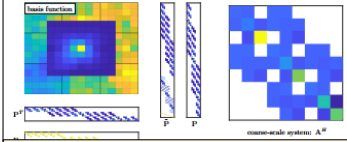
New book: Advanced modelling with MRST



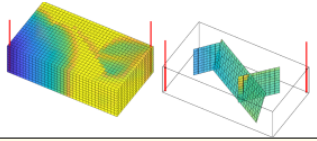
Berge et al.: Constrained Voronoi grids



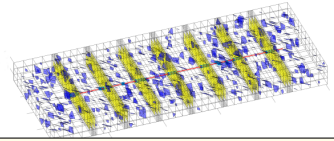
Al Kobaisi & Zhang: nonlinear FVM



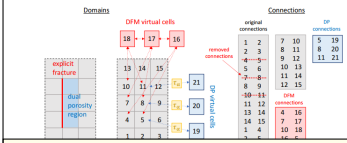
Lie & Møyner: multiscale methods



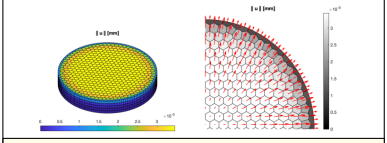
Wong et al.: embedded discrete fractures



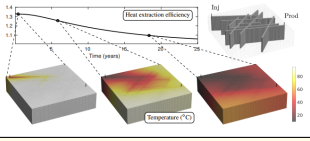
Olorode et al.; fractured unconventionals



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

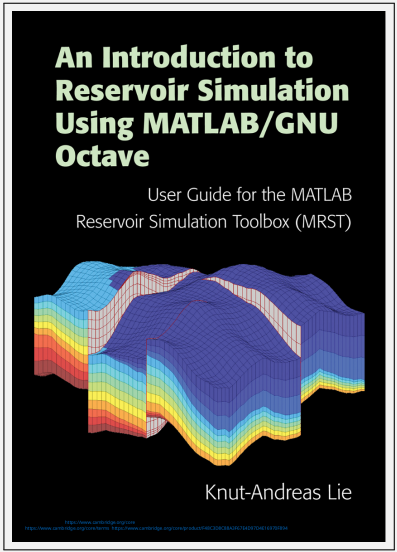
Møyner: compositional

Andersen: coupled flow & geomechanics

MRST Textbook
Knut-Andreas Lie. *An Introduction to Reservoir Simulation Using Matlab/GNU octave*. Cambridge University Press, 2019 (Open access). DOI: 10.1017/9781108591416

Download, documentation, tutorials and more
MRST website: mrst.no
Repositories: bitbucket.org/mrst/mrst-core/wiki

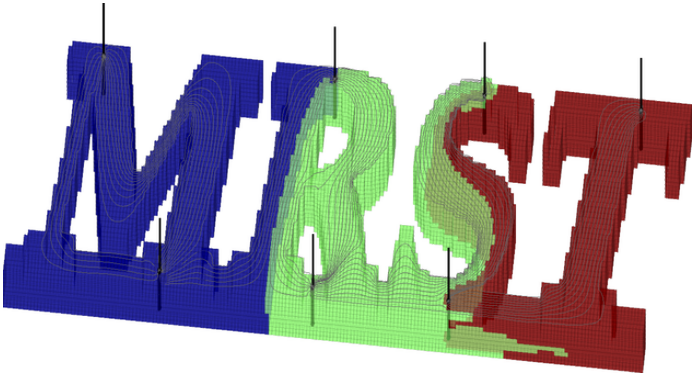
Tutorial lectures (including full-length version of this talk)
mrst.no/documentation/tutorial-lectures/



Acknowledgements

Thanks to all co-developers at SINTEF, 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, ...