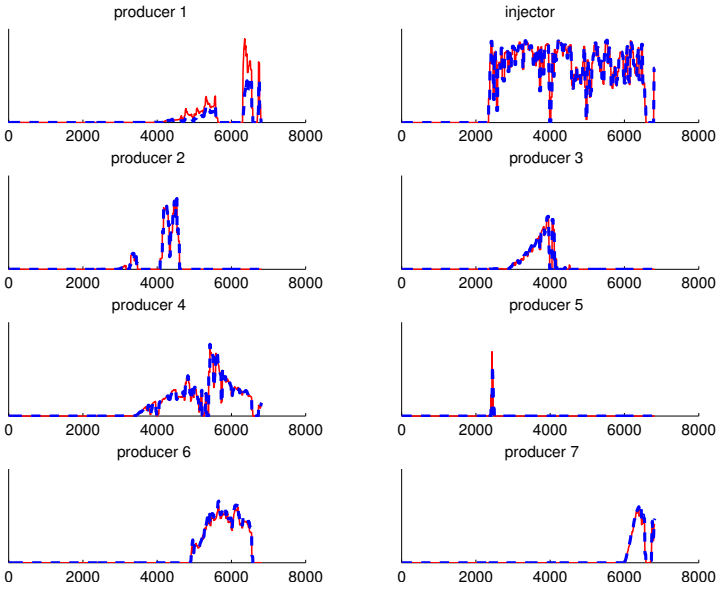
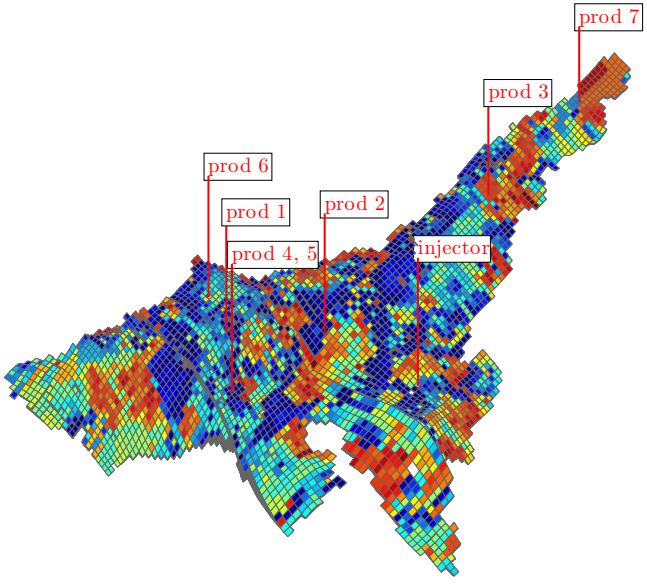
Rapid Prototyping with the Matlab Reservoir Simulation Toolbox (MRST)

Knut–Andreas Lie, SINTEF, Oslo, Norway

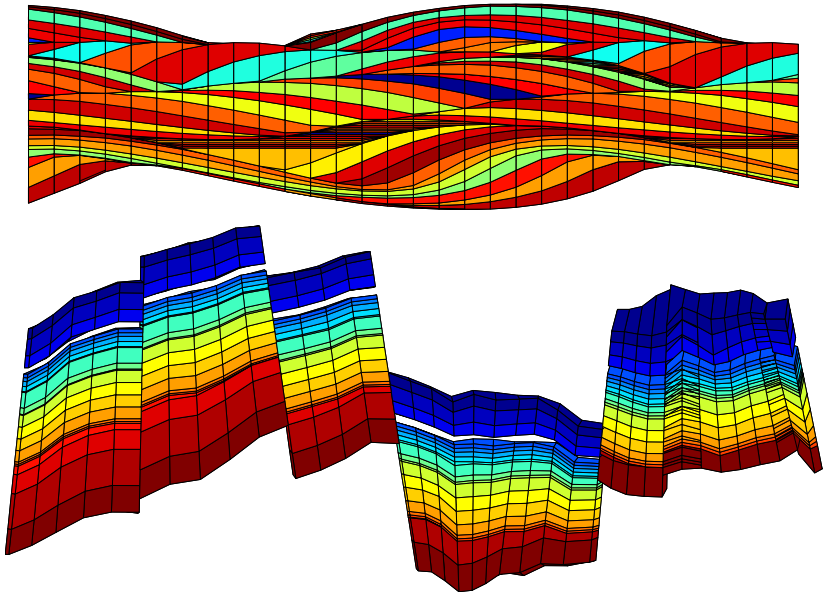
Darsim Lecture Series, TU Delft - 2 October 2015

Setting: reservoir simulation





- Complex unstructured grids, orders of magnitude parameter variations
- Flow models: system of PDEs with elliptic and hyperbolic sub-character
- Well models: analytic sub-models
- Sensitivities and gradients for optimization, etc
- New methods $\longrightarrow$ experimental programming



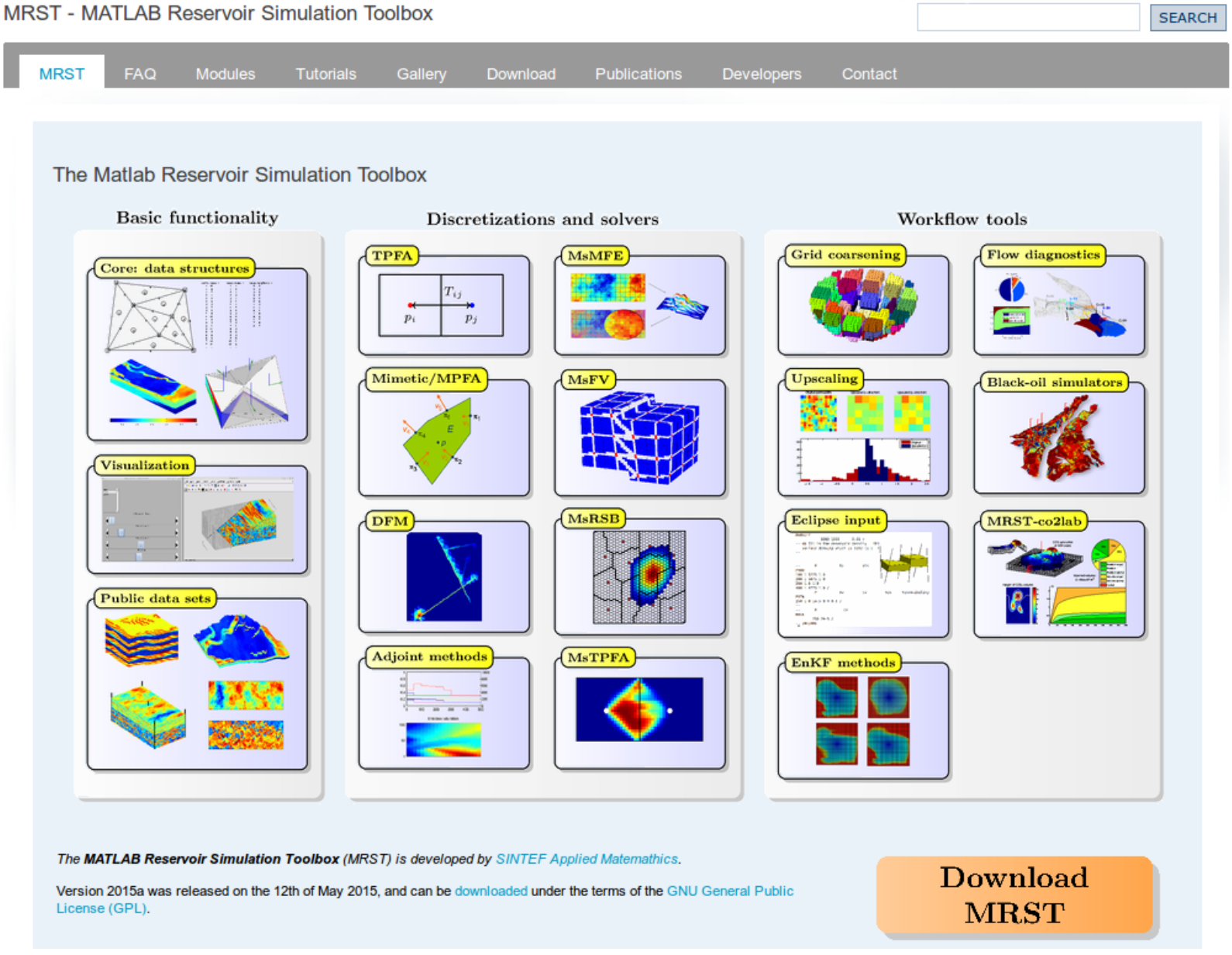
Choice of language for experimental programming

If your thoughts are complex and constantly developing, you need the flexibility to make the road as you go.

	3rd generation Fortran, C, C++	4th generation Matlab, Python
Syntax	complicated	intuitive
Cross-platform	challenging	✓
Build process	complicated	✗
Linking of external libraries	usually a mess	✗
Type-checking	static	dynamic
Mathematical abstractions	user-defined	built-in
Numerical computations	libraries	built-in
Data analysis and visualization	libraries/external	built-in
Debugger, profiling, etc	external/IDE	built-in
Traversing data structures	loops, iterators	vectorization†

† also: indirection maps and logical indices

Accelerating the development cycle: MRST



http://www.sintef.no/MRST

Accelerating the development cycle: MRST

Originally:

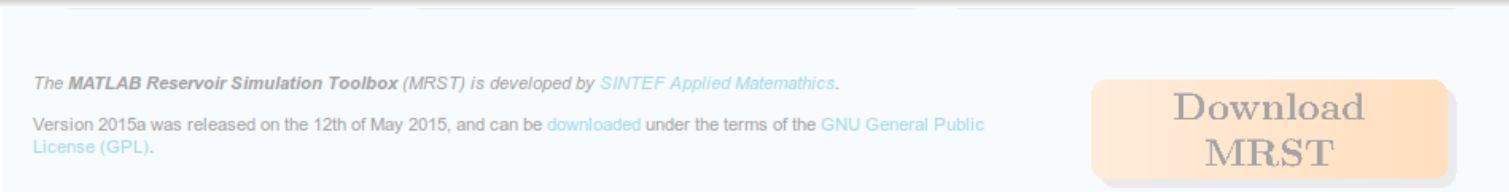
- developed to support research on multiscale methods and mimetic discretizations
- first public release as open source, April 2009

Today:

- general toolbox for rapid prototyping and verification of new computational methods
- wide range of applications
- two releases per year, current: 2015a, May 2015
- past four releases: a thousand unique downloads each

Users:

- academic institutions, research institutes, oil and service companies
- large user base in USA, Norway, China, Brazil, United Kingdom, Iran, Germany, Netherlands, France, Canada, ...



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

Key ideas for rapid prototyping

Focus on clean and simple implementation of equations, close to the mathematics $\longrightarrow$ less error-prone coding, simpler to maintain and extend

- **hide specific details** of grid, rock, discretization, and constitutive laws
- **unstructured grid format** – algorithms can be implemented without knowing the specifics of the grid
- **vectorize** – no visible loops and few indices, i.e., 1-to-1 correspondence between continuous and discrete variables
- **discrete operators**, mappings, and forms – not tied to specific flow equations and can be precomputed independently
  also: seamless change of discretization scheme
- **fluid objects** – unified access to standard quantities
  desired: auto-generated from input decks, but simple to modify/extend
- **automatic differentiation** – no need to derive gradients and Jacobians analytically

Discrete operators div and grad

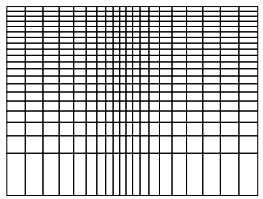
Discrete divergence is a mapping: $\mathbb{R}^{n_f} \to \mathbb{R}^{n_c}$

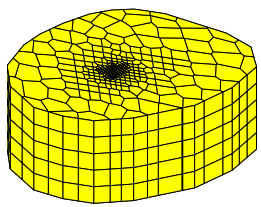
$$\texttt{div}(\boldsymbol{v})[c] = \sum_{f \in F(c)} \boldsymbol{v}[f]\, \mathbf{1}_{\{c=N_1(f)\}} - \sum_{f \in F(c)} \boldsymbol{v}[f]\, \mathbf{1}_{\{c=N_2(f)\}}$$

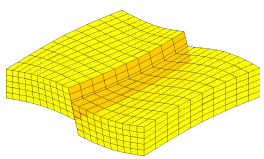
Discrete gradient is a mapping: $\mathbb{R}^{n_c} \to \mathbb{R}^{n_f}$

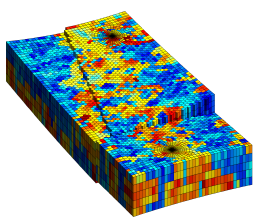
$$\texttt{grad}(\boldsymbol{p})[f] = \boldsymbol{p}[N_2(f)] - \boldsymbol{p}[N_1(f)]$$

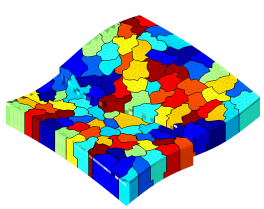
For no-flow boundary problems, we have $\texttt{div} = -\texttt{grad}^T$











Discrete operators div and grad

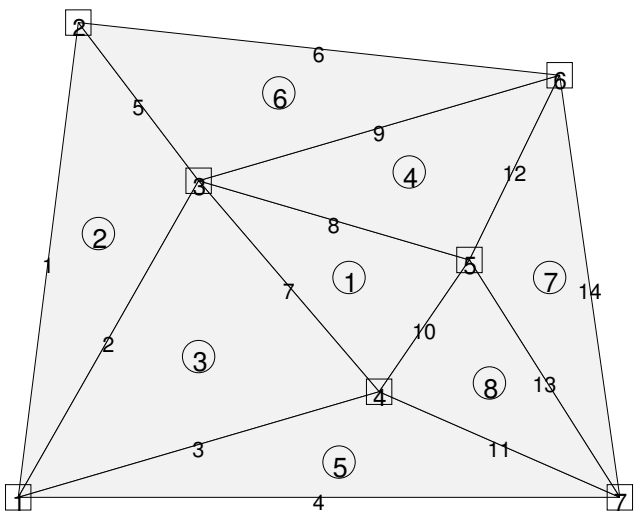
Discrete divergence is a mapping: $\mathbb{R}^{n_f} \to \mathbb{R}^{n_c}$

$$\texttt{div}(\boldsymbol{v})[c] = \sum_{f \in F(c)} \boldsymbol{v}[f]\, \mathbf{1}_{\{c=N_1(f)\}} - \sum_{f \in F(c)} \boldsymbol{v}[f]\, \mathbf{1}_{\{c=N_2(f)\}}$$

Discrete gradient is a mapping: $\mathbb{R}^{n_c} \to \mathbb{R}^{n_f}$

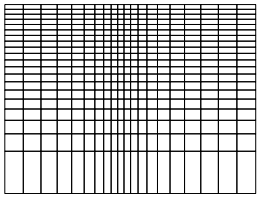
$$\texttt{grad}(\boldsymbol{p})[f] = \boldsymbol{p}[N_2(f)] - \boldsymbol{p}[N_1(f)]$$

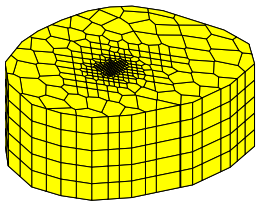
For no-flow boundary problems, we have $\texttt{div} = -\texttt{grad}^T$

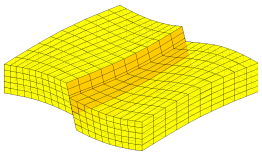


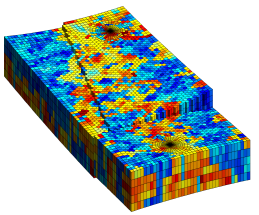
cells.faces =		faces.nodes =		faces.neighbors =	
1	10	1	1	0	2
1	8	1	2	2	3
1	7	2	1	3	5
2	1	2	3	5	0
2	2	3	1	6	2
2	5	3	4	0	6
3	3	4	1	1	3
3	7	4	7	4	1
3	2	5	2	6	4
4	8	5	3	1	8
4	12	6	2	8	5
4	9	6	6	4	7
5	3	7	3	7	8
5	4	7	4	0	7
5	11	8	3		
6	9	8	5		
6	6	9	3		
6	5	9	6		
7	13	10	4		
7	14	10	5		
:	:	:	:		

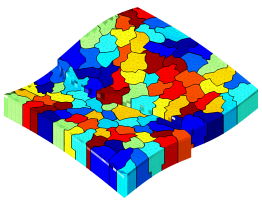
$F(c)$ $N_1(f)$ $N_2(f)$











Automatic differentiation

- Techniques for simultaneously computing and keeping track of values and derivatives
- Any code can be broken down to a limited set of arithmetic operations and elementary functions (`sin`, `exp`, `power`, ...).
- Operator overloading is used to implement common derivative rules

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

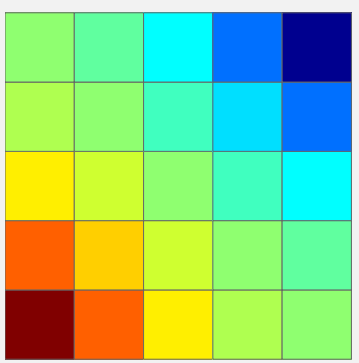
```
x = ADI Properties:        y = ADI Properties:        z = ADI Properties:
    val: 1                     val: 2                     val: 0.4060
    jac: {[1]  [0]}            jac: {[0]  [1]}            jac: {[-0.8120]  [-0.4060]}
```

$\frac{\partial x}{\partial x}$ $\frac{\partial x}{\partial y}$ $\frac{\partial y}{\partial x}$ $\frac{\partial y}{\partial y}$ $\frac{\partial z}{\partial x}\Big|_{x=1,y=2}$ $\frac{\partial z}{\partial y}\Big|_{x=1,y=2}$

Solving the Poisson equation: $-\Delta p = q$

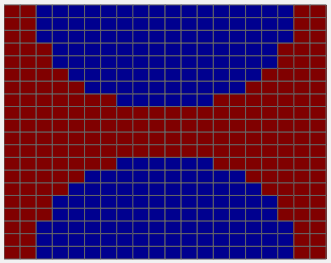
```
% Grid and grid information
G = cartGrid([5 5]);
N = G.faces.neighbors;
N = N(all(N ~= 0, 2), :);
nf = size(N,1);
nc = G.cells.num;

% Operators
C = sparse([(1:nf)'; (1:nf)'], N, ...
           ones(nf,1)*[-1 1], nf, nc);
grad = @(x) C*x;
div  = @(x) -C'*x;

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

eq    = div(grad(p))+q;          % equation
eq(1) = eq(1) + p(1);            % make solution unique
p     = -eq.jac{1}\eq.val;       % solve equation
plotCellData(G,p);
```

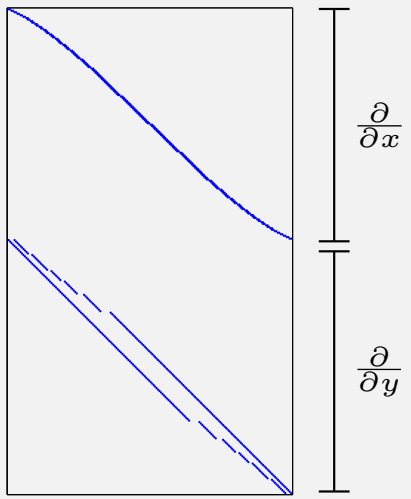
$$C = \begin{bmatrix} \\ \\ \\ \\ \\ \\ \\ \\ \end{bmatrix} \quad \frac{\partial}{\partial x} \quad \frac{\partial}{\partial y}$$

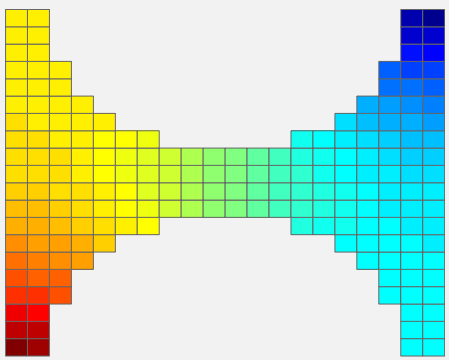


Solving the Poisson equation: non-rectangular domain

```
% Grid
G = cartGrid([20 20],[1  1]);
G = computeGeometry(G);
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));

% Grid information
N = G.faces.neighbors;
:
% Operators
C = sparse([(1:nf)'; (1:nf)'], N, ...
           ones(nf,1)*[-1 1], nf, nc);
:
% Assemble and solve equations
p = initVariablesADI(zeros(nc,1));
q = zeros(nc, 1);
q(1) = 1; q(nc) = -1;

eq    = div(grad(p))+q;
eq(1) = eq(1) + p(1);
p     = -eq.jac{1}\eq.val;
plotCellData(G,p);
```







Single-phase incompressible flow: unstructured grid

Fundamental physics: Darcy's law

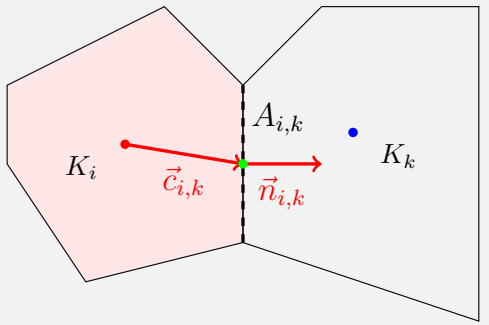
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

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

Conservation of mass:

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

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



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

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

Single-phase incompressible flow: unstructured grid

Fundamental physics: Darcy's law

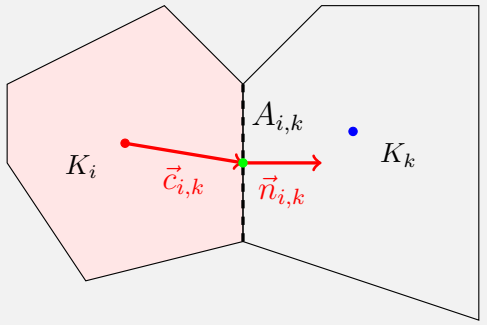
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

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

Conservation of mass:

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

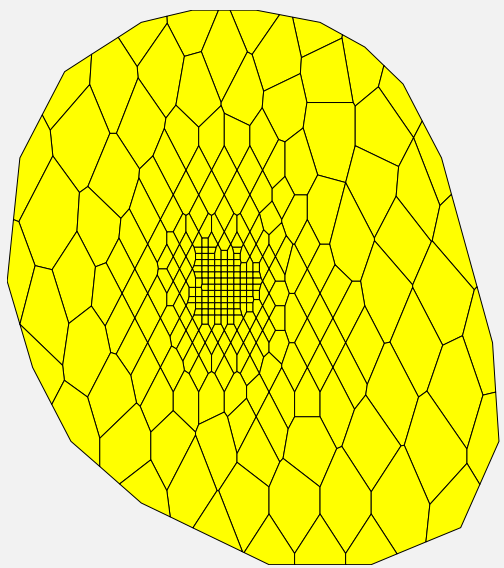
$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



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

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

```
load seamount
G = pebi(triangleGrid([x(:) y(:)], delaunay(x,y)));
G = computeGeometry(G);
:
```



Single-phase incompressible flow: unstructured grid

Fundamental physics: Darcy's law

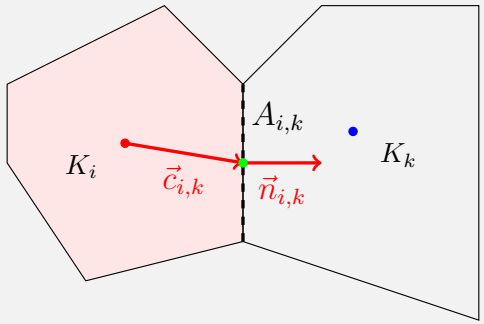
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

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

Conservation of mass:

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

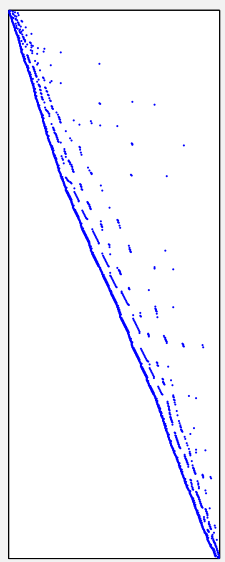
$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



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

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

```
load seamount
G = pebi(triangleGrid([x(:) y(:)], delaunay(x,y)));
G = computeGeometry(G);
:
```



Single-phase incompressible flow: unstructured grid

Fundamental physics: Darcy's law

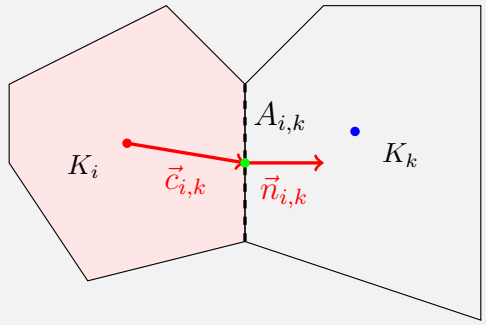
$$\int_{\Gamma_f} \vec{v}(x) \cdot \vec{n}_f \, ds = -\int_{\Gamma_f} \mathbf{K}(x) \nabla p \cdot \vec{n}_f \, ds$$

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

Conservation of mass:

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

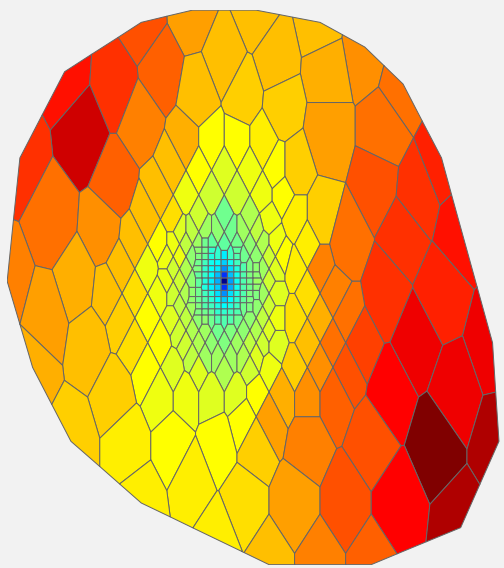
$$\mathtt{div}(\boldsymbol{v})[c] = \boldsymbol{q}[c]$$



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

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

```
load seamount
G = pebi(triangleGrid([x(:) y(:)], delaunay(x,y)));
G = computeGeometry(G);
:
hT = computeTrans(G, struct('perm', ones(nc,1)*darcy));
cf = G.cells.faces(:,1);
T  = 1 ./ accumarray(cf, 1 ./ hT, [G.faces.num, 1]);
T  = T(all(N~=0,2),:);
:
q([135 282 17]) = [-1 .5 .5];
eq    = div(T.*grad(p))+q;
eq(1) = eq(1) + p(1);
p     = -eq.jac{1}\eq.val;
```



Single-phase weakly compressible flow

The governing equation is

$$c\frac{\partial p}{\partial t} - \operatorname{div}(\mathbf{K}\operatorname{grad} p) = 0$$

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

$$\frac{1}{\Delta t}c(p^{n+1} - p^n) + \texttt{div}(\rho v)^{n+1} = q, \quad v = -\frac{K}{\mu}\texttt{grad}(p)$$

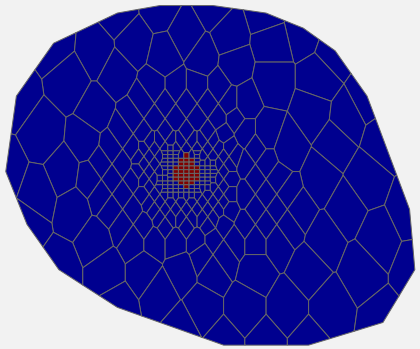
```
presEq = @(p, p0, dt)     (1/dt)*c*(p-p0) - div( (T/mu).*grad(p));
```

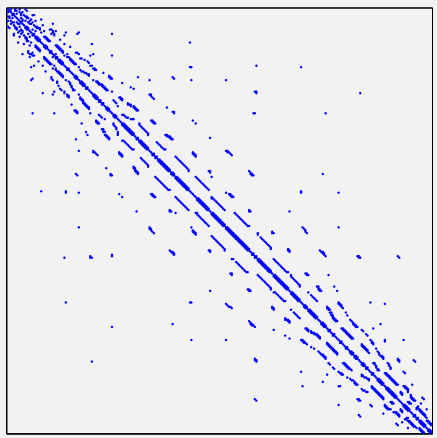
current time step

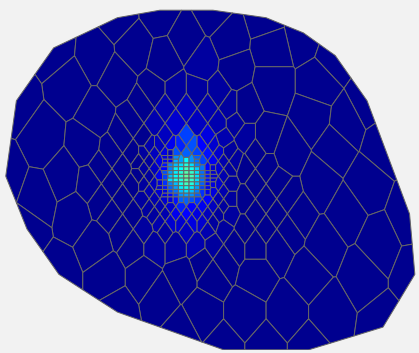
previous time step

Single-phase weakly compressible flow

```
G.nodes.coords = G.nodes.coords*100;
:
c = 1e-4;
mu = 1*centi*poise;

p0 = 100*atm*ones(nc, 1);
p0(r<5) = 200*atm;

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

[t,T,dt] = deal(0,10*day,hour);
p = initVariablesADI(p0);
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 solver

$$c\frac{\partial p}{\partial t} - \operatorname{div}(\mathbf{K}\operatorname{grad} p) = 0$$

```
%  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));
```

rock and fluid compressible solver

$$\frac{\partial}{\partial t}(\phi\rho) - \operatorname{div}(\rho\mathbf{K}\operatorname{grad} p) = 0$$

```
% Rock property
phi0 = 0.3; c_r = 1e-3; pr = 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: $\mathbb{R}^{nc} \rightarrow \mathbb{R}^{nf}$

Single-phase compressible flow: adding gravity

Semi-discrete flow equations on residual form:

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

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

Single-phase compressible flow: adding well model

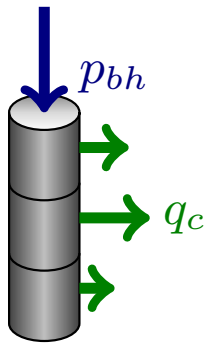
Peaceman well model:

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

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

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

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



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;
```

Single-phase compressible flow: time loop

```
[p, bhp, qS] = ...
    initVariablesADI(p_init, p_init(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
```

Single-phase compressible flow: time loop

```
[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
```

Rapid prototyping: thermal effects

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

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

Constitutive laws and operators

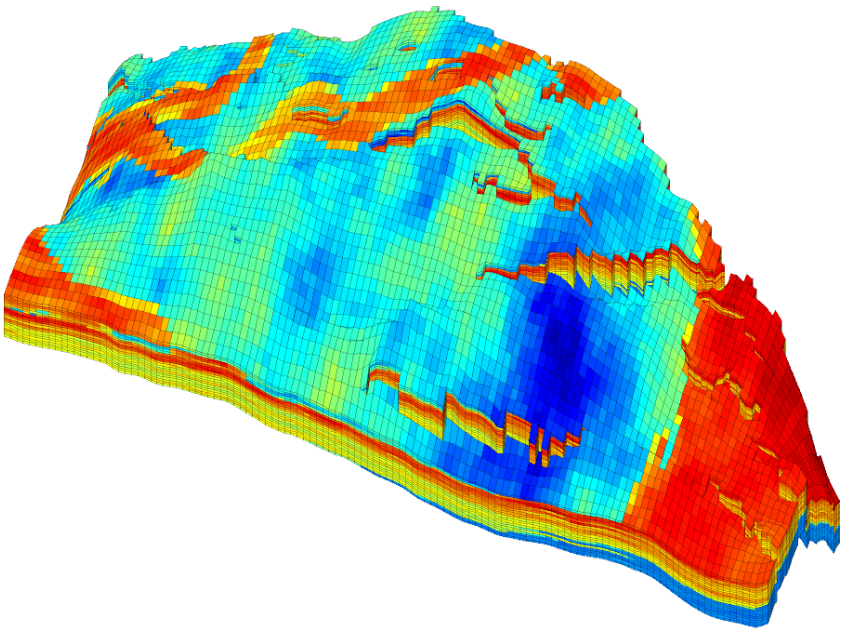
```
pvr  = poreVolume(G, rock);
pv   = @(p) pvr .* exp( cr * (p - pr) );
spv  = @(p) G.cells.volumes - pv(p);
:
rho  = @(p,T) rhor.*(1+(cp*(p - pr))).*exp(-ct*(T-Tr));
mu   = @(p,T) mu0*(1+cmup*(p-p_r)).*exp(-cmut*(T-T_r));
:
Hf   = @(p,T) Cw*T + (1-Tr*ct).*(p-pr)./rho(p,T);
Ef   = @(p,T) Hf(p,T) - p./rho(p,T);
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));
```

Full realism: corner-point grids from Eclipse

```
% Load data and look at keywords
mrstModule add gui;
mrstDatasetGUI;      % download SAIGUP model
grdecl = readGRDECL(fullfile('data', 'SAIGUP', 'SAIGUP.GRDECL'));

% Convert to SI units
usys   = getUnitSystem('METRIC');
grdecl = convertInputUnits(grdecl, usys);
G = processGRDECL(grdecl, 'Verbose', true);
G = computeGeometry(G);
rock = grdecl2Rock(grdecl, G.cells.indexMap);
plotCellData(G, log10(rock.perm(:,1)));
```



Recap: 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
  - wide range of built-in functions for numerical computations
  - powerful data analysis, graphical user interface, and visualization
- 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
- Later, one can, if necessary, replace bottleneck operations with accelerated editions implemented in compiled language

Also: use scripting language as a wrapper when you develop solvers in compiled languages

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

Next step: object-orientation

Code starting to become complicated:

- wells with advanced schedules and controls
- time-step control and iteration control
- CPR type preconditioners and multigrid solvers
- hysteretic behavior (post-iteration updates)
- nested iterations (non-Newtonian fluids)
- advanced flow models that are extension of simpler models
- sub-equations with different discretizations, ...

Introduce object-orientation to separate:

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

and only expose needed details and enable more reuse of previous functionality

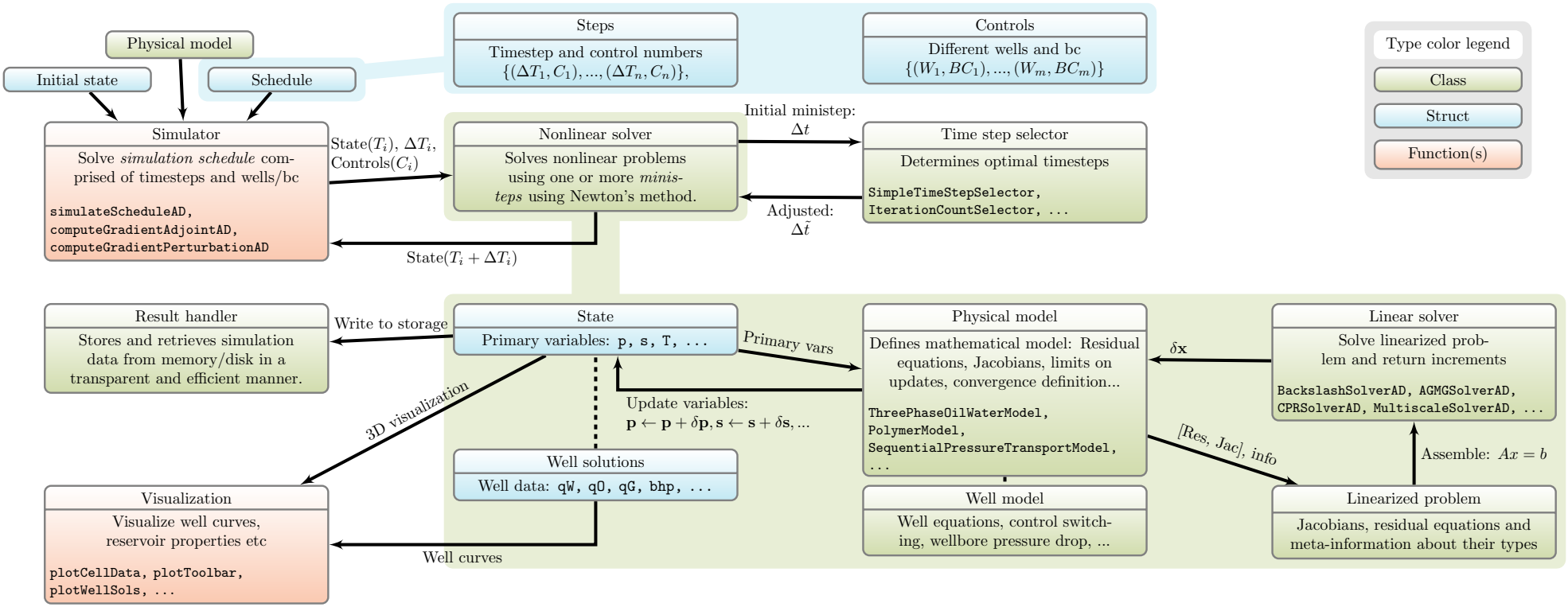
Advanced simulators: Motivation

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

- standardized interfaces make Newton solver independent of physical model
- standardized input/output makes it easy to compare and plot results
- switching linear solvers or time-stepping strategy is straightforward
  In 2015b release: examples of both fully implicit and sequential solvers)

Typical workflow: Build simple prototype $\rightarrow$ migrate to class-based solver

The layout of the AD solvers



You 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.

The black-oil model

Model equations:

$$\partial_t(\phi b_o s_o) + \nabla \cdot (b_o v_o) - b_o q_o = 0$$
$$\partial_t(\phi b_w s_w) + \nabla \cdot (b_w v_w) - b_w q_w = 0$$
$$\partial_t[\phi(b_g s_g + b_o r_s s_o)] + \nabla \cdot (b_g v_g + b_o r_s v_o) - (b_g q_g + b_o r_s q_o) = 0$$
$$v_j = -\frac{k_{rj}}{\mu_j} K(\nabla p_j - \rho_h g \nabla z)$$

Water equation discetized in time:

$$\frac{V}{\Delta t}(\phi^{n+1} b_w^{n+1} s_w^{n+1} - \phi^n b_w^n s_w^n) + \nabla \cdot (b_w^{n+1} v_w^{n+1}) = 0$$

Water equation in Matlab code using autodiff:

```
eqs{2} = (s.pv/dt).*(pvMult.*bW.*sW - pvMult0.*f.bW(p0).*sW0) ...
         + s.div(bWvW);
```

Black-oil model: water equation in more detail

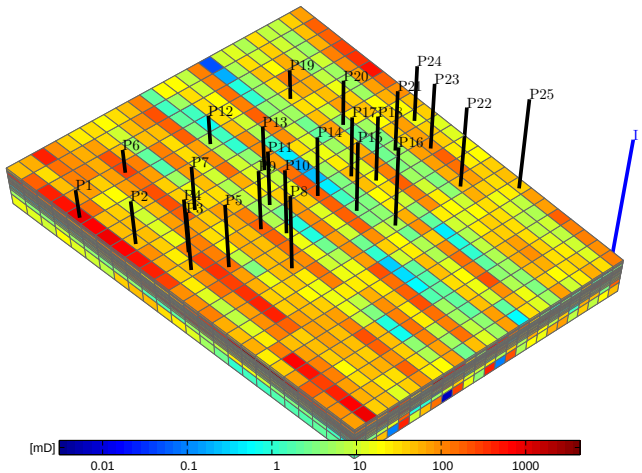
```
function [eqs,..] = eqsfiOW(state0, state, ..)
p  = state.pressure;   sW = state.s(:,1);      % unknowns
p0 = state0.pressure; sW0 = state0.s(:,1);     % previous step
:
% water properties (evaluated using oil pressure)
[krW, krO, krG] = f.relPerm(sW);
bW   = f.bW(p);
rhoW = bW .* f.rhoWS;
rhoWf= S.faceAvg(rhoW);
mobW = f.tranMultR(p) .* krW ./ f.muW(p);

% upstream weighting
gdz  = S.Grad(G.cells.centroids)*grav';
dpW  = S.Grad(p - f.pcOW(sW)) - rhoWf.*gdz;
upc  = (double(dpW) <= 0);
bWvW = -S.faceUpstr(upc, bW.*mobW) .* S.T .* dpW;

% discrete equation
eq{2} = S.Div(bWvW) + (S.pv/dt) ...
        .*(f.pvMultR(p).*bW.*sW - f.pvMult(p0).*f.bW(p0).*sW0 );
```

— discrete operators — fluid object

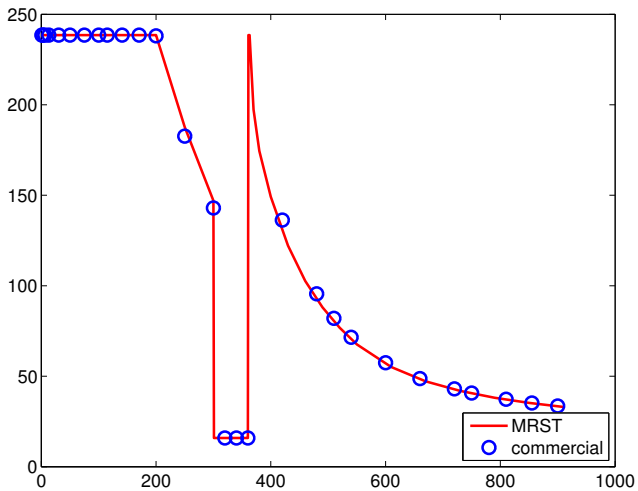
Example: SPE9 3-phase benchmark

- Grid with 9000 cells
- 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
- Almost perfect match between MRST and a commercial simulator

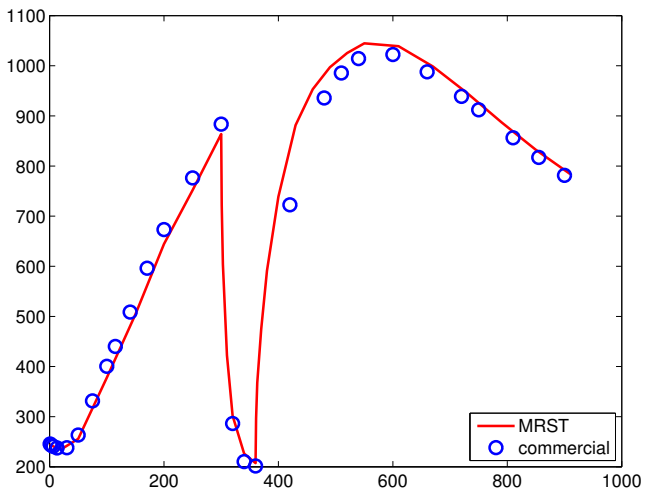
From: modules/ad-blackoil/examples/spe9/



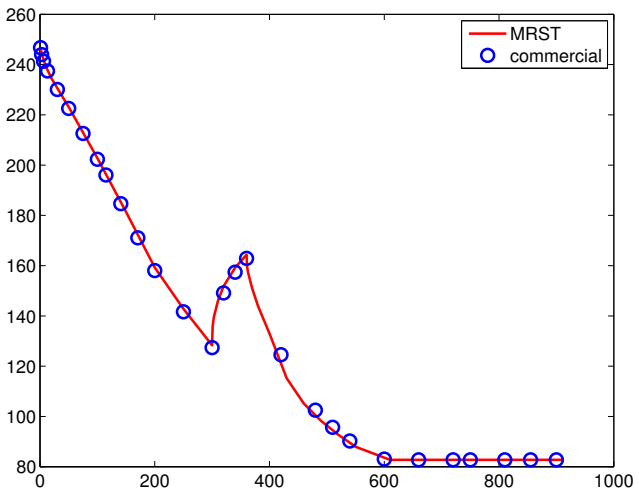
Oil rate (P2)



Gas-oil ratio (P5)

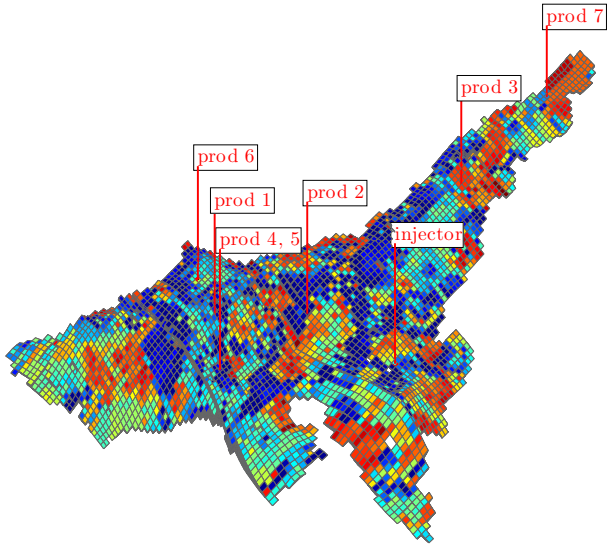


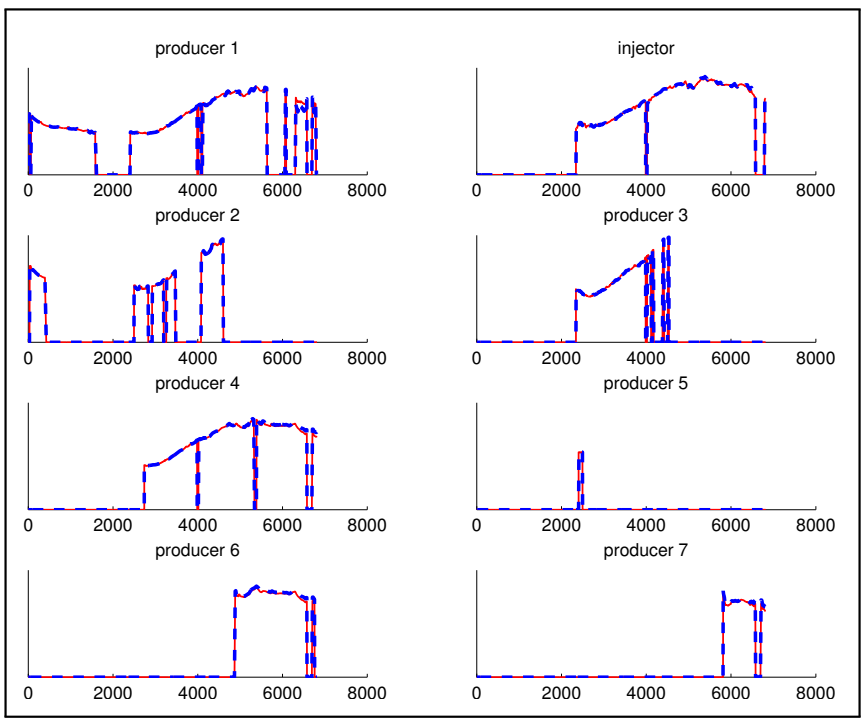
Bottom-hole pressure (P4)



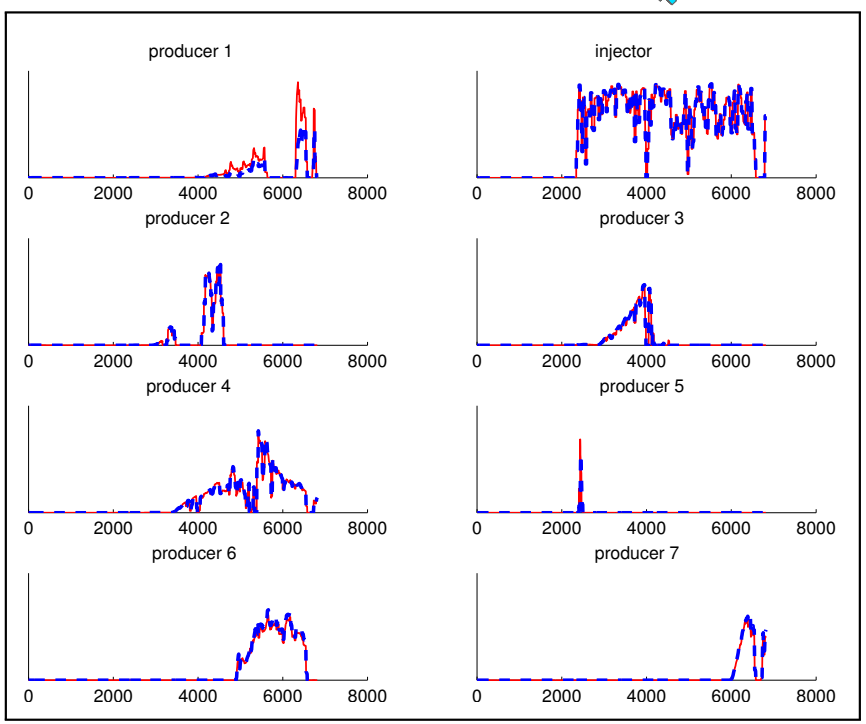
Example: Voador field (Petrobras)

- South wing of the reservoir
- Gradients obtained through adjoint simulations
- Benchmarked against commercial simulator:
  - 20 years of historic data
  - virtually identical results
  - main challenge: needed to reverse-engineer description of wells. . .





bottom-hole pressure



water rate

Rapid prototyping: polymer model

From: modules/ad-blackoil/utils/equationsOilWater.m

```
function [eqs,..] = eqsfiOW(state0, state, ..)
p  = state.pressure;   sW = state.s(:,1);      % unknowns
p0 = state0.pressure; sW0 = state0.s(:,1);    % previous step
:
% water properties (evaluated using oil pressure)
[krW, krO, krG] = f.relPerm(sW);
bW   = f.bW(p);
rhoW = bW .* f.rhoWS;
rhoWf= S.faceAvg(rhoW);
mobW = f.tranMultR(p) .* krW ./ f.muW(p);

% upstream weighting
gdz  = S.Grad(G.cells.centroids)*grav';
dpW  = S.Grad(p - f.pcOW(sW)) - rhoWf.*gdz;
upc  = (double(dpW) <= 0);
bWvW = -S.faceUpstr(upc, bW.*mobW) .* S.T .* dpW;

% discrete equation
eq{2} = S.Div(bWvW) + (S.pv/dt) ...
        .*(f.pvMultR(p).*bW.*sW - f.pvMult(p0).*f.bW(p0).*sW0 );
```

Add polymer model with:

- Todd–Longstaff mixing
- polymer adsorption
- permeability reduction (hysteresis)

(for pedagogical purposes: no shear effects, etc)

Rapid prototyping: polymer model

From: modules/ad-blackoil/utils/equationsOilWater.m

```
function [eqs,..] = eqsfiOW(state0, state, ..)
p  = state.pressure;   sW = state.s(:,1);      % unknowns
p0 = state0.pressure; sW0 = state0.s(:,1);     % previous step
:
% water properties (evaluated using oil pressure)
[krW, krO, krG] = f.relPerm(sW);
bW   = f.bW(p);
rhoW = bW .* f.rhoWS;
rhoWf= S.faceAvg(rhoW);
mobW = f.tranMultR(p) .* krW ./ f.muW(p);

% upstream weighting
gdz  = S.Grad(G.cells.centroids)*grav';
dpW  = S.Grad(p - f.pcOW(sW)) - rhoWf.*gdz;
upc  = (double(dpW) <= 0);
bWvW = -S.faceUpstr(upc, bW.*mobW) .* S.T .* dpW;

% discrete equation
eq{2} = S.Div(bWvW) + (S.pv/dt) ...
        .*(f.pvMultR(p).*bW.*sW - f.pvMult(p0).*f.bW(p0).*
```

Add polymer model with:

- Todd–Longstaff mixing
- polymer adsorption
- permeability reduction (hysteresis)

(for pedagogical purposes: no shear effects, etc)

```
% polymer viscosity modifier
cbar = c/f.cmax;
a    = f.muWMult(f.cmax).^(1-mixpar);
b    = 1./(1-cbar+cbar./a);
permRed = 1 + ((f.rrf-1)./f.adsMax).*effads(c, cmax, f);
muWmult = b.*permRed.*f.muWMult(c).^mixpar;

% water properties
[krW, krO, krG] = f.relPerm(sW);
:
muW  = f.muW(p);
mobW = f.tranMultR(p) .* krW ./(muWmult*muW);

% polymer properties
mobP = (mobW.*c)./(a + (1-a)*cbar);

% upstream weighting
:
bWvW = -S.faceUpstr(upc, bW.*mobW) .* S.T .* dpW;
bWvp = -S.faceUpstr(upc, bW.*mobP) .* S.T .* dpW;

% discrete equation for aqua phase
eq{2} = S.Div(bWvP) + (S.pv/dt) ...
        .*(f.pvMultR(p).*bW.*sW - f.pvMult(p0).*f.bW(p0).*sW0);

% discrete equation for polymer component
eq{3} = S.Div(bWvW) + (S.pv/dt) .* (...
      f.pvMultR(p).*bW.*sW.*Cp - f.pvMult(p0).*f.bW(p0).*sW0.*Cp0 ...
      + f.rhoR.*((1-poro)./poro).*...
        ( effads(c, cmax, f) - effads(c0, cmax0, f) ) );
```

Rapid prototyping: polymer model class

Oil water model

```
classdef OWModel < ReservoirModel
methods
function model = OWModel(G, rock, fluid)
    model = model@ReservoirModel(G, rock, fluid);
    model.oil = true;
    model.gas = false;
    model.water = true;
    model.saturationVarNames = {'sw', 'so'};
    model = model.setupOperators(G, rock);
end

function [problem, state] = getEquations(..)
    [problem, state] = equationsOilWater(..);
end

function [fn, index] = getVariableField(model, name)
    switch(lower(name))
        case {'sw', 'so'}
            index = model.satVarIndex(name);
            fn = 's';
        case {'s', 'saturation'}
            index = 1:numel(model.saturationVarNames);
            fn = 's';
        case {'pressure', 'p'}
            index = 1;
            fn = 'pressure';
        ...
    end
end
end
```

Modified model

```
classdef PolymerOWModel < OWModel
methods
...
function [problem, state] = getEquations(..)
    [problem, state] = equationsOilWaterPolymer(..);
end

function [state, report] = updateState(..)
    [state, report] = updateState@OWModel(..);
    c = model.getProp(state, 'polymer');
    c = min(c, model.fluid.cmax);
    state = model.setProp(state, 'c', max(c, 0));
end

function [fn, index] = getVariableField(model, name)
    switch(lower(name))
        case 'c'
            index = 1;
            fn = 'c';
        case 'cmax'
            index = 1;
            fn = 'cmax';
        otherwise
            [fn, index] = ...
            getVariableField@OWModel(model, name);
    end
end

function state = updateAfterConvergence(state, varargin)
    % Hysteresis-like polymer effect
    state = updateAfterConvergence@OWModel(state);
    [c, cmax] = model.getProp(state, 'c', 'cmax');
    state = model.setProp(state, 'cmax', max(cmax, c));
end
```

Concluding remarks – prototyping

This combination of technologies has been a big success for our group

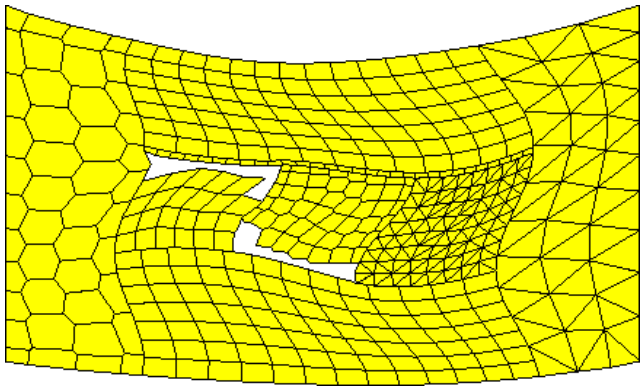
Main benefits of framework

- simplifies own research and makes team members more productive
- forces internal collaboration across projects
- enables us to validate new methods on industry-standard problems and verify against commercial simulators
- accelerated our development of software tools in complied languages

Research: geomechanics

Aims:

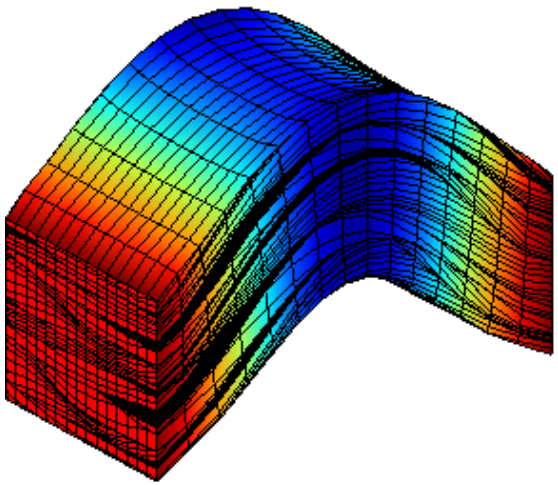
- mechanics on geological models without regridding.
- simulation of hydraulic fracturing and fault activation



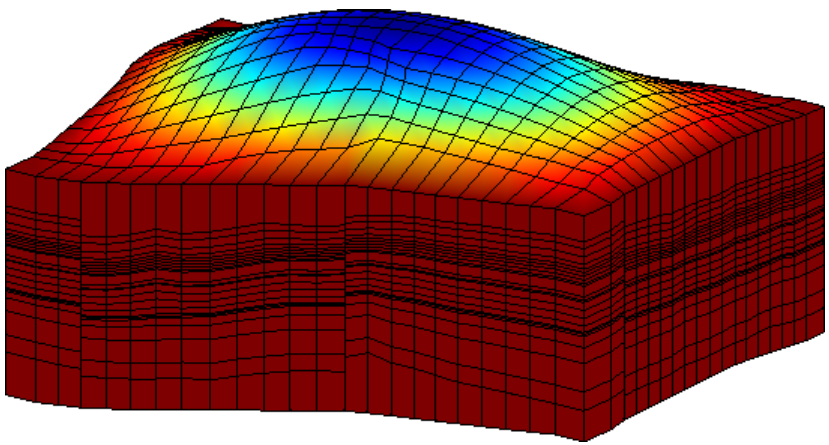
VEM on polygonal 2D grid

Methods for elasticity on polyhedral cells:

- Virtual element methods (VEM): finite element for polyhedrals
- Multi-point stress approximation (MPSA): multipoint flux approximation for elasticity
- Mimetic method: mixed method for polyhedrals

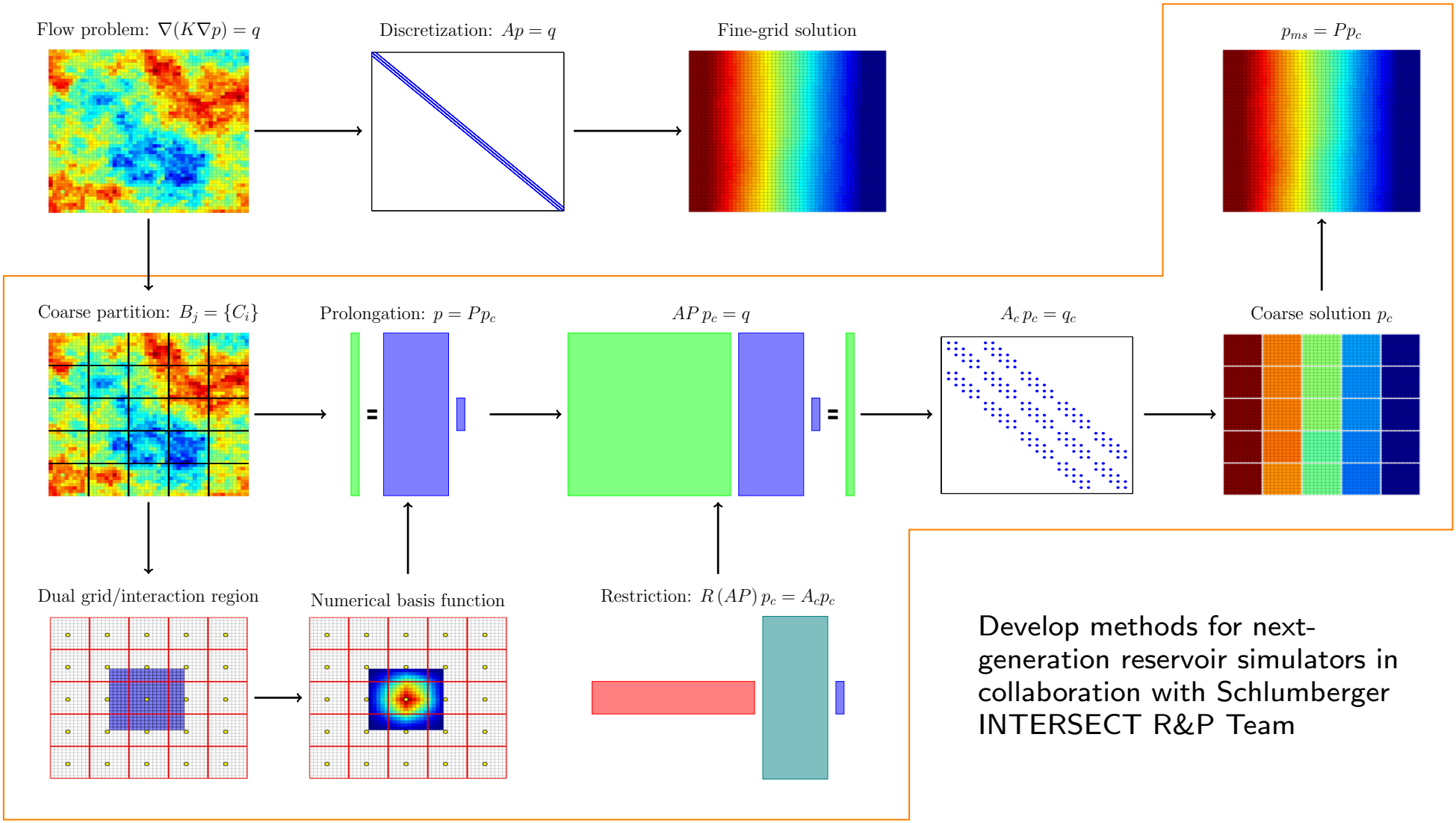


VEM on SBED model



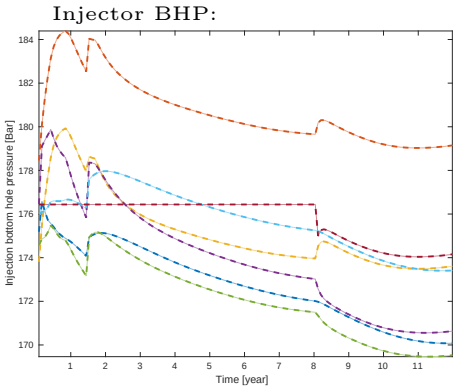
VEM on model embedding corner-point grid

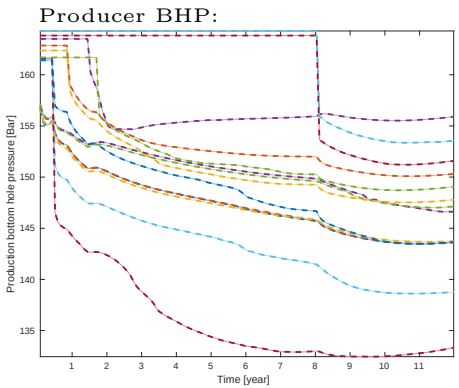
Research: multiscale methods



Develop methods for next-generation reservoir simulators in collaboration with Schlumberger INTERSECT R&P Team

Example: applicability to real field models

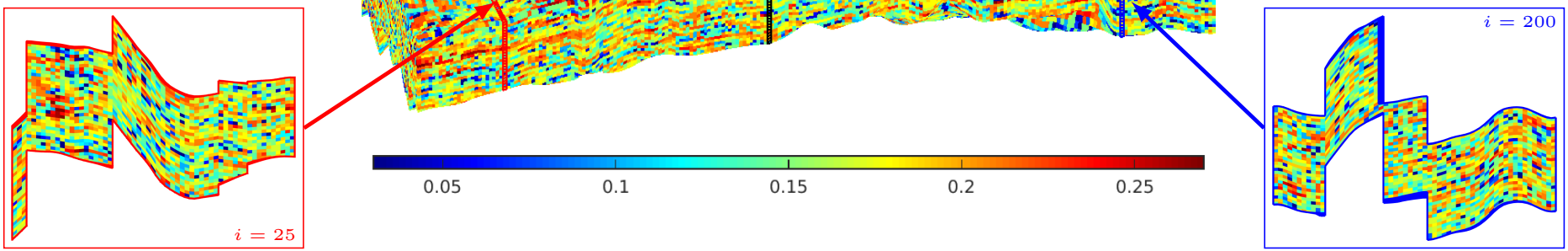




Watt Field: water flooding

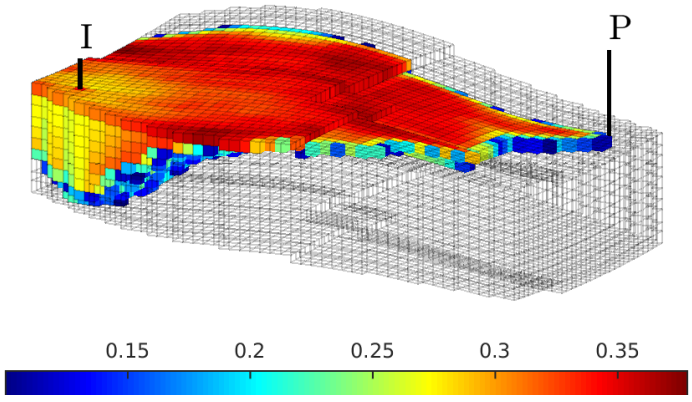
415 711 active cells, three rock types

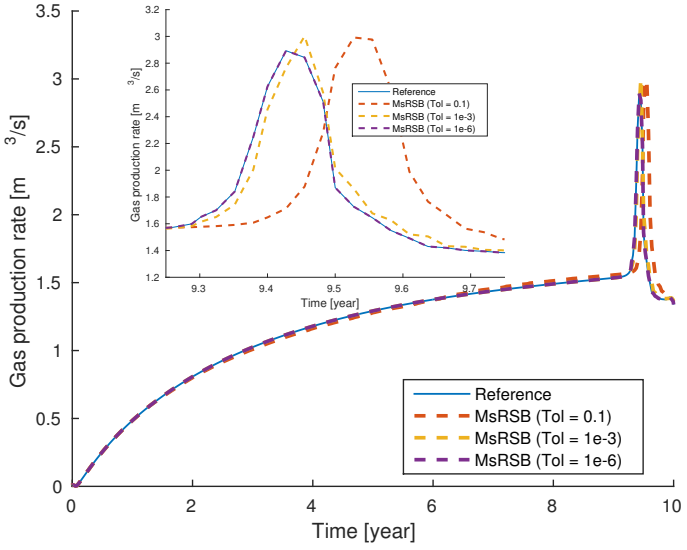
7 injectors, 15 horizontal producers



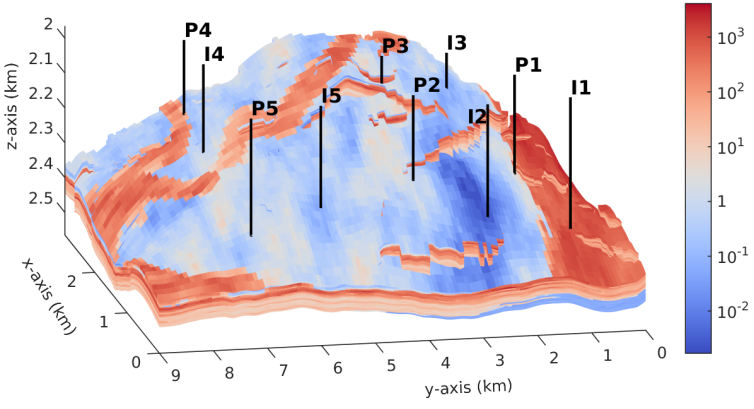
Example: extension to realistic flow physics

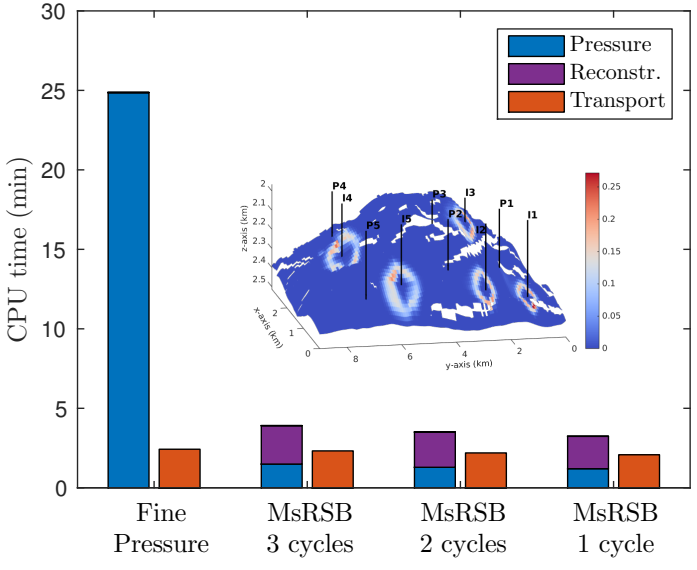
3-phase black-oil: solution gas



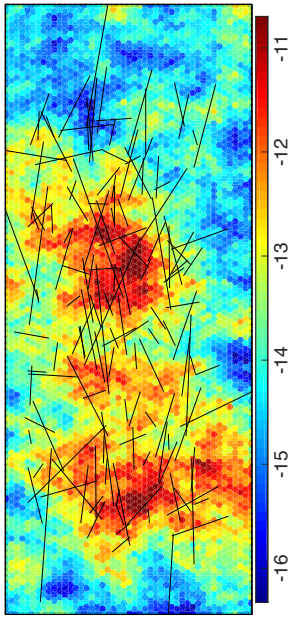


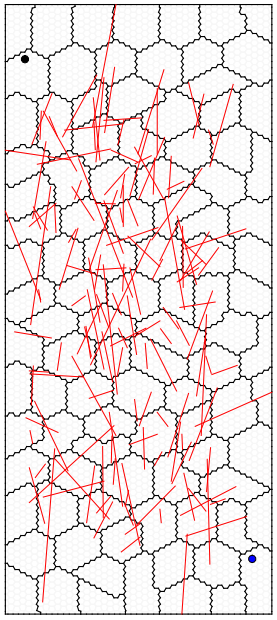
EOR: polymer injection w/shear effects

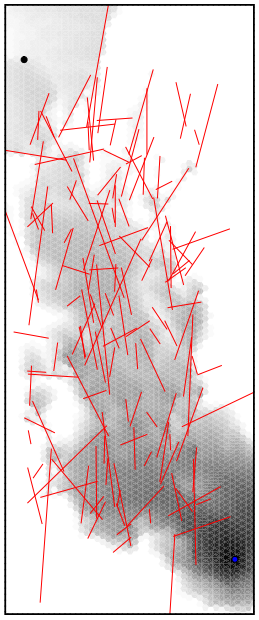


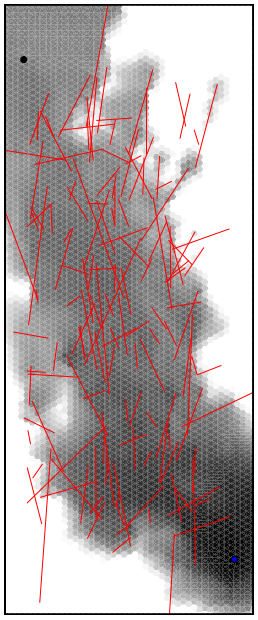


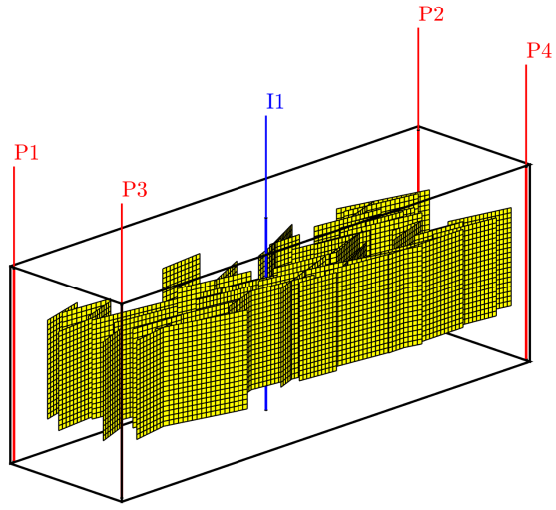
Example: hierarchical fracture modeling

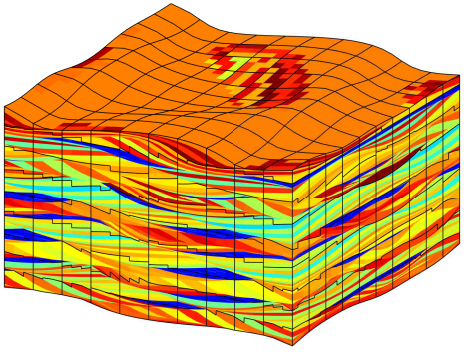


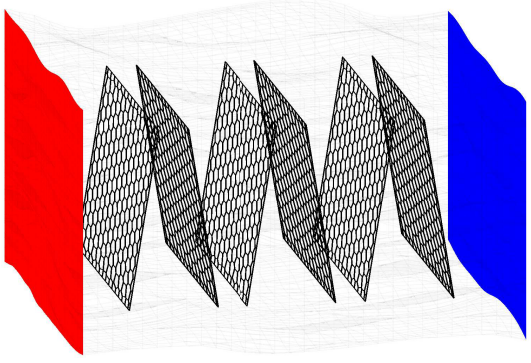


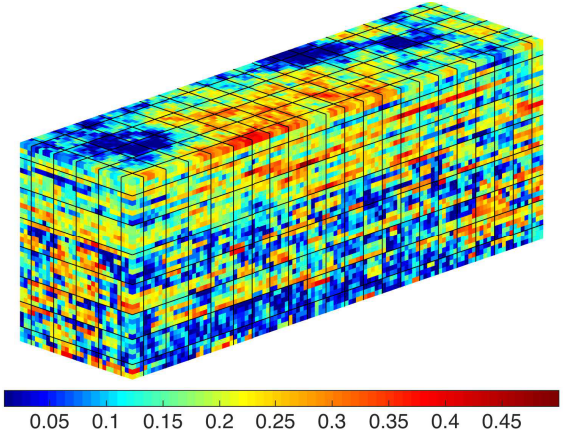






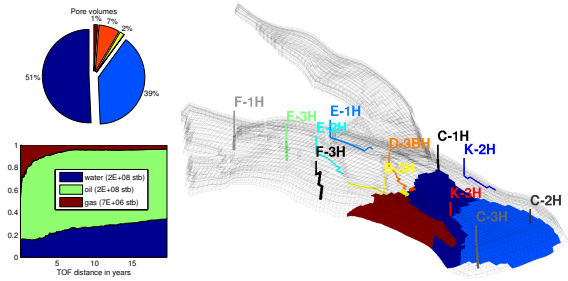




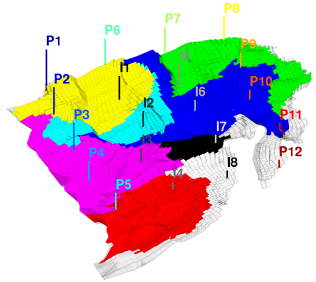


Collaboration with H. Hajibeygi, S. Shah, and M. Tene. (TU Delft)

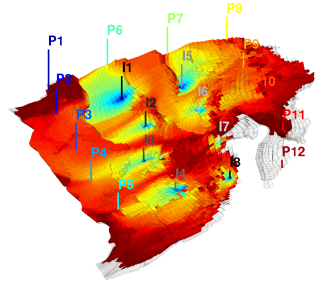
Visualisation



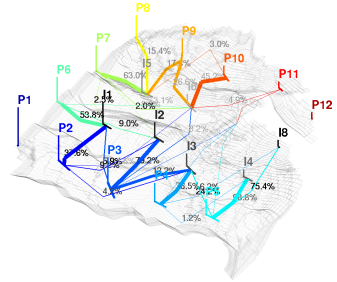
Delineation of drainage volume



Sweep volumes for injectors



Time-of-flight from injectors

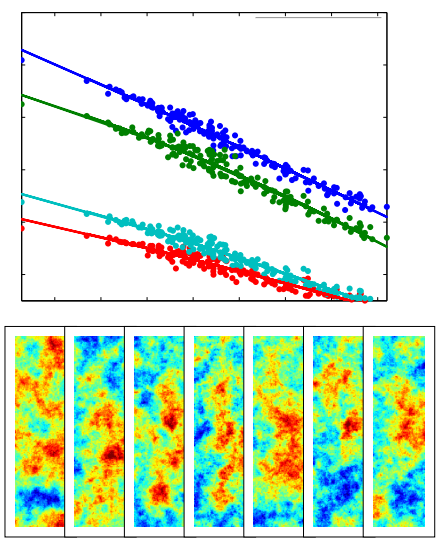


Flow allocation for well pairs

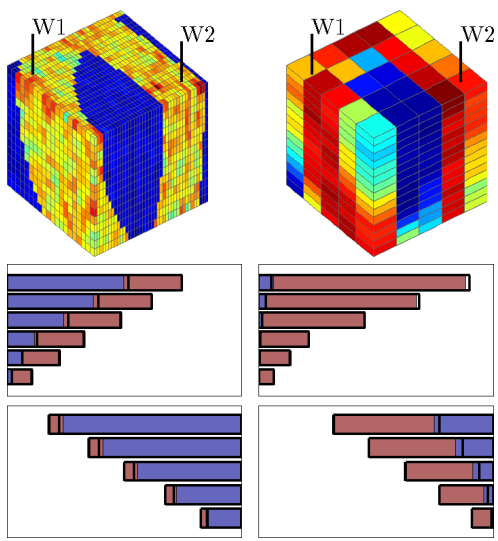
Flow diagnostics

Simple, controlled numerical experiments run to probe a reservoir model and establish connections and basic volume estimates. Can also be used to quickly compute flow proxies.

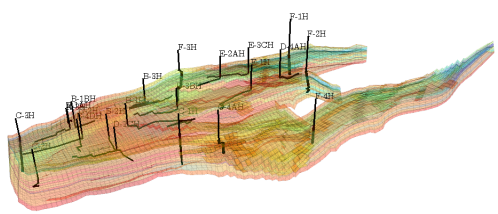
Ranking



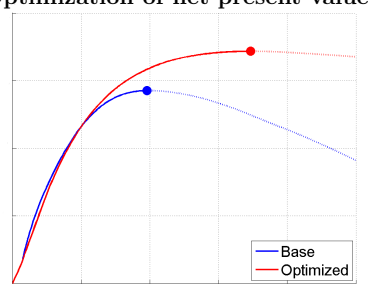
Verify upscaling



Optimisation

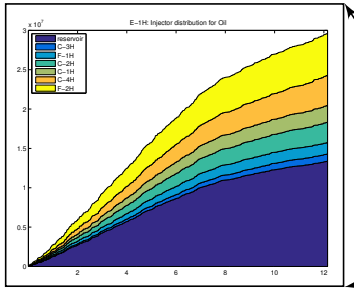


Optimization of net present value

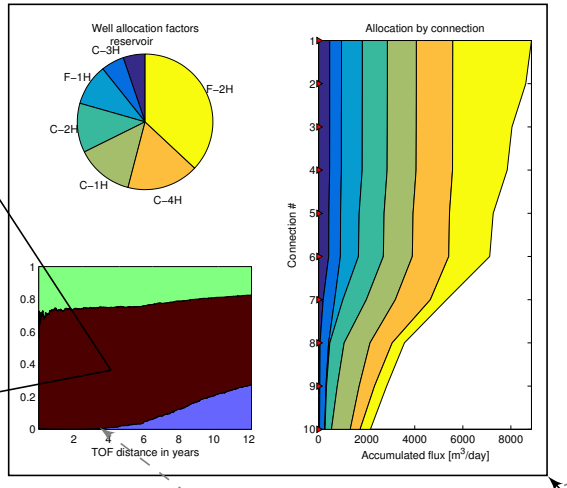


Example: enhanced interactive visualization

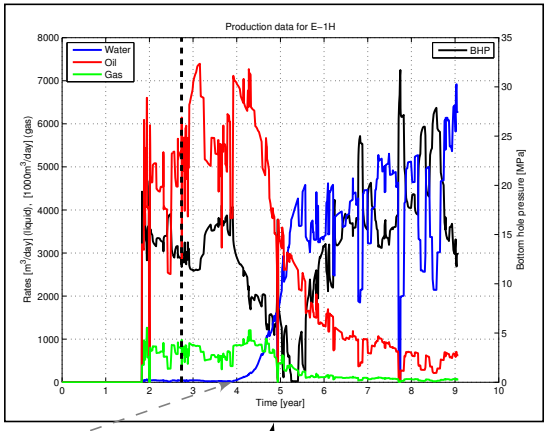
Injectors and fluid expansion contributing to oil production



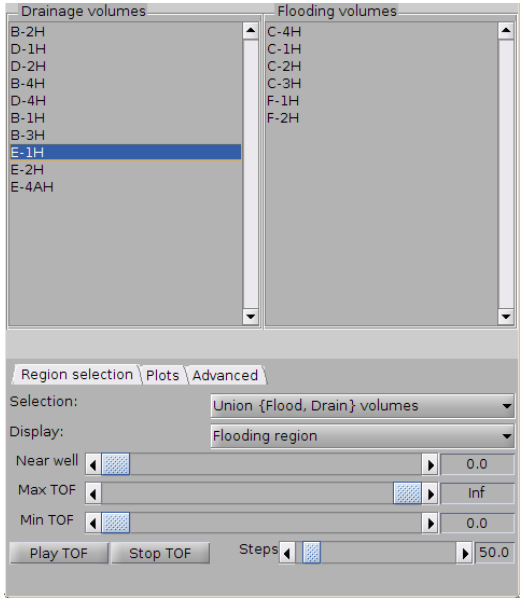
Volumetrics for drainage zone of E-1H

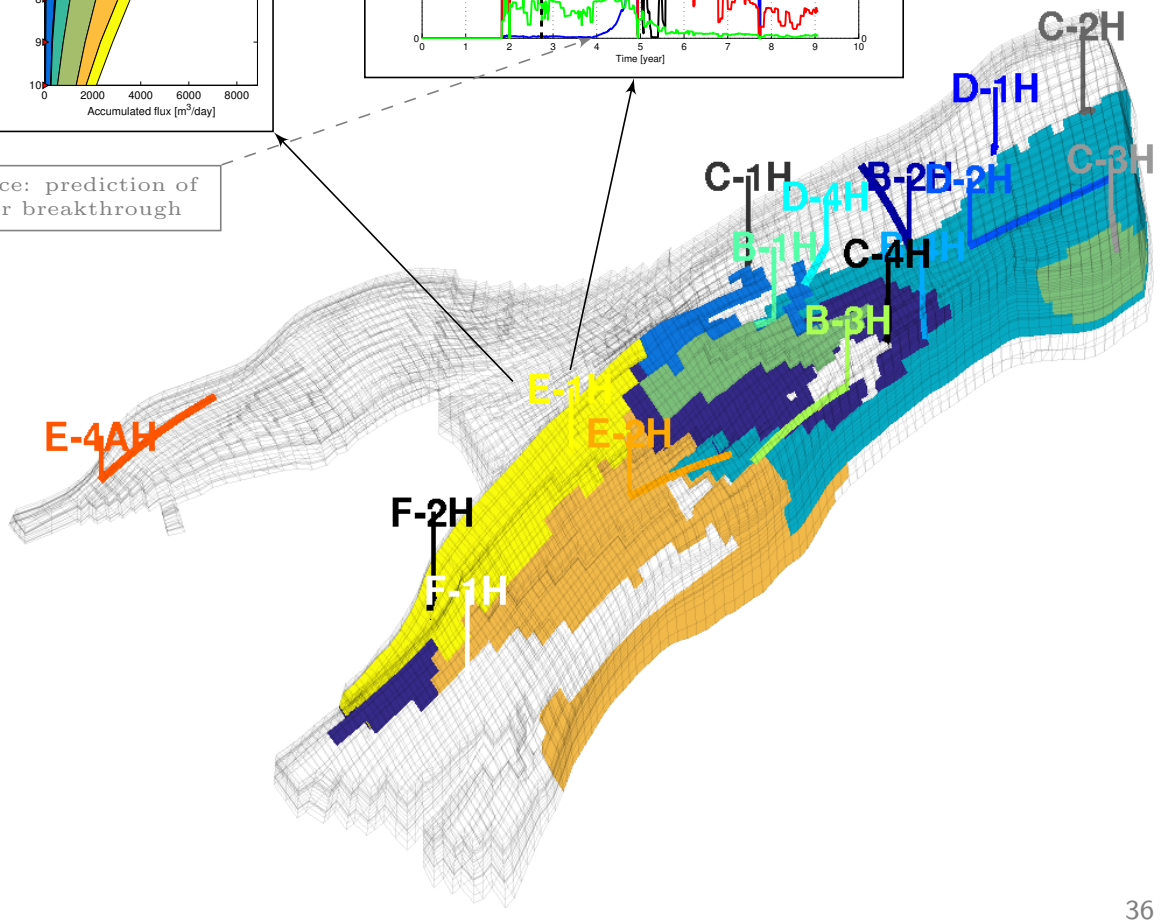


Simulated production for E-1H



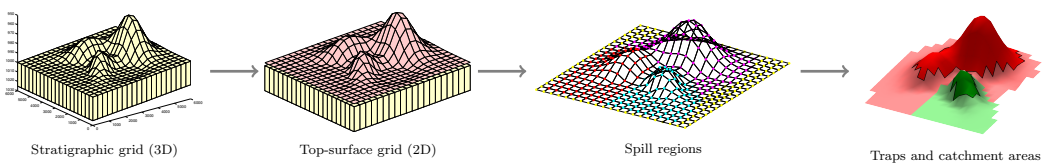
Notice: prediction of water breakthrough



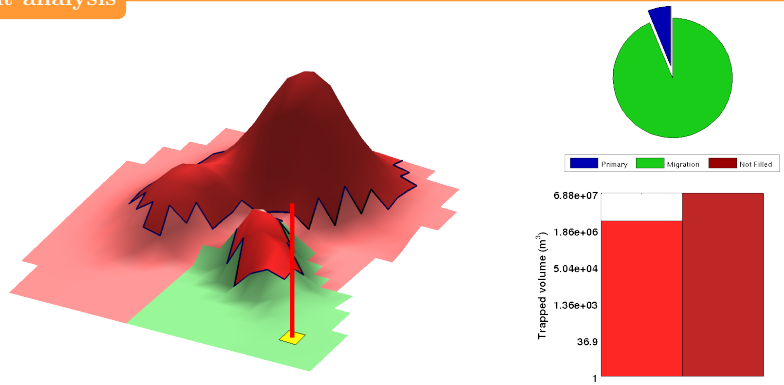


Research: geological CO_2 storage

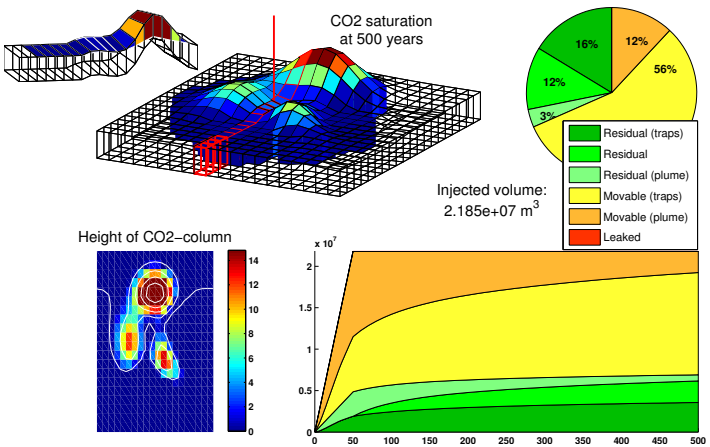
Traps and catchment areas

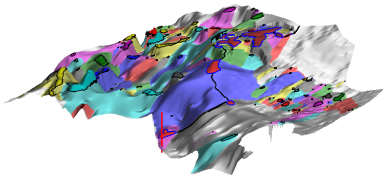


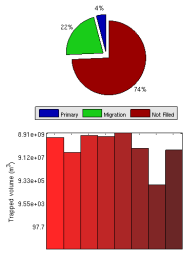
Spill-point analysis

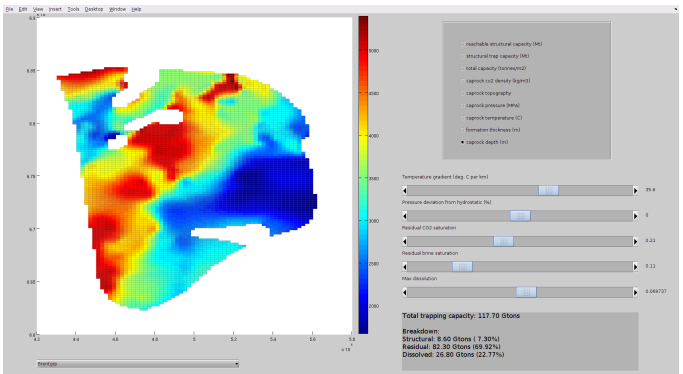


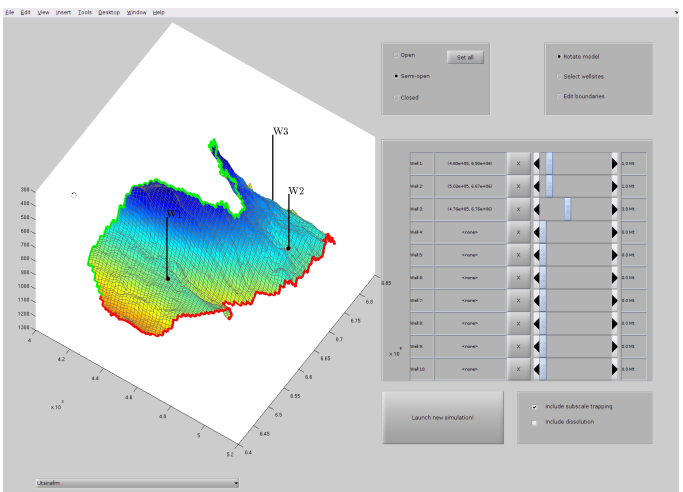
VE simulation



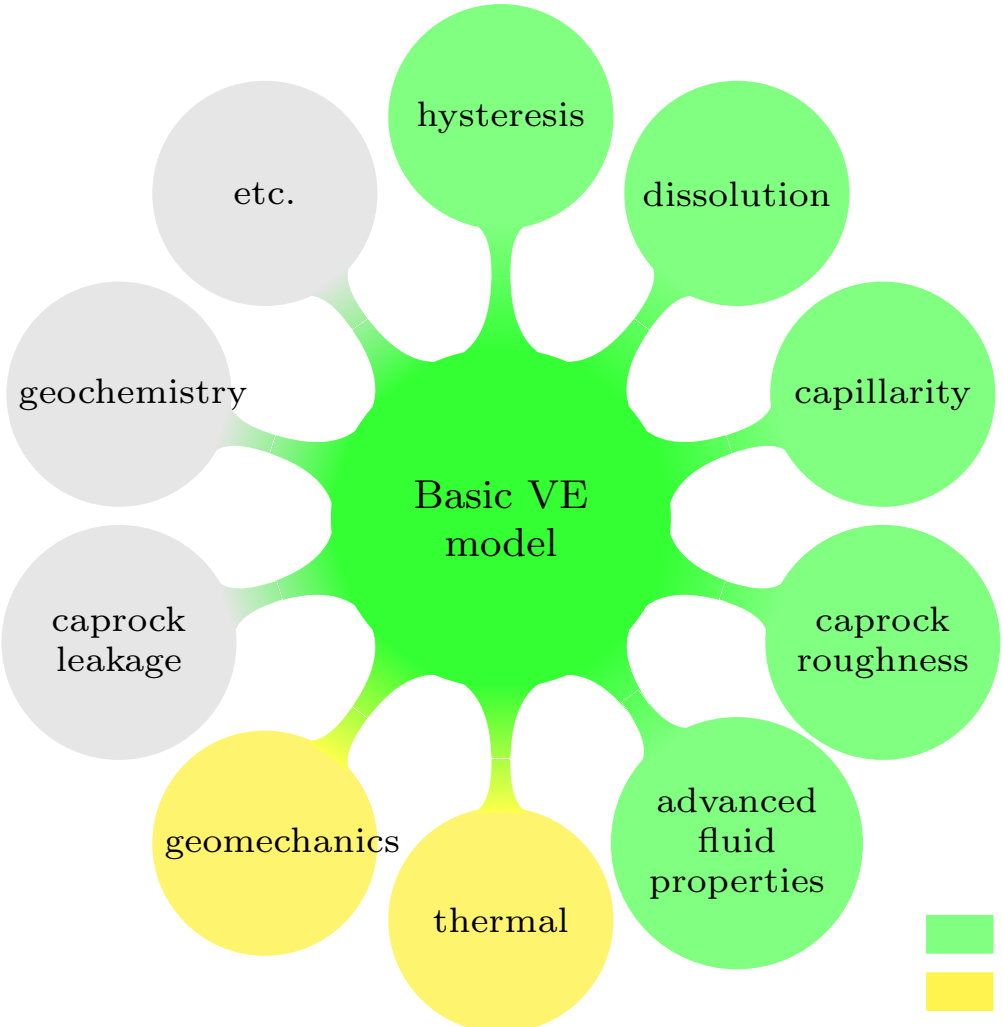


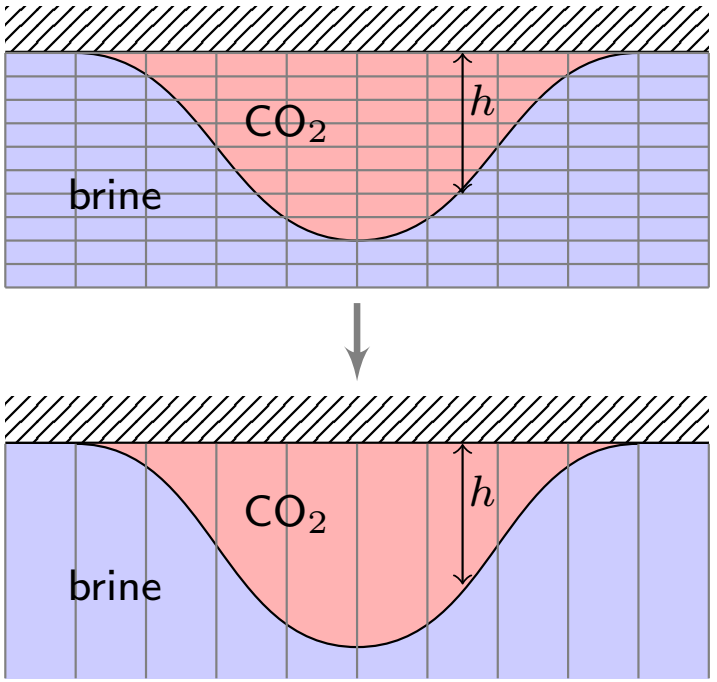




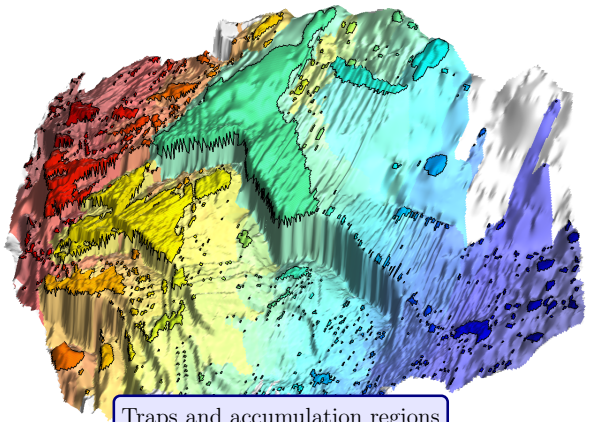


Research: vertical equilibrium models

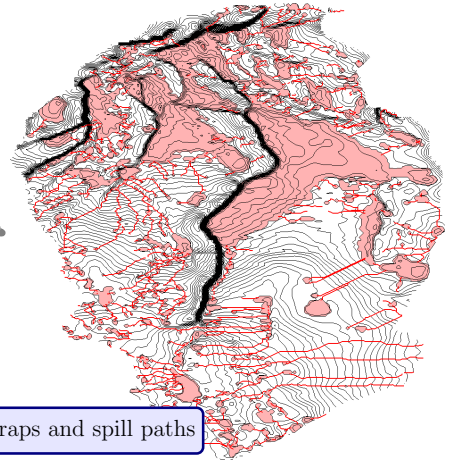




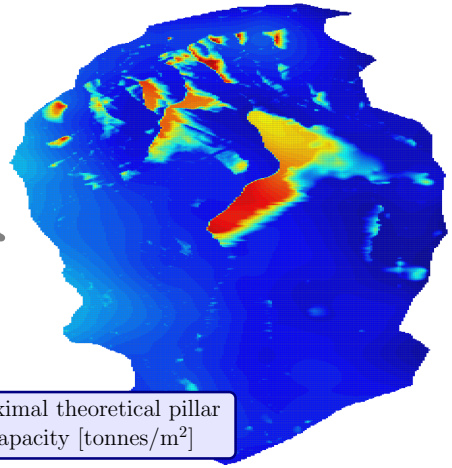
Example: Johansen formation



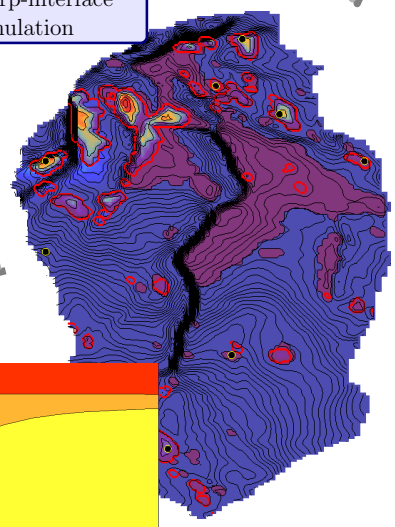
Traps and accumulation regions



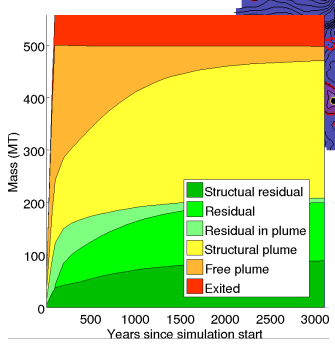
Traps and spill paths

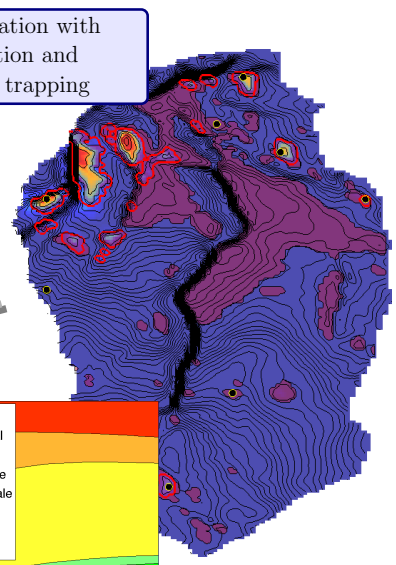


Maximal theoretical pillar capacity [tonnes/m^2]

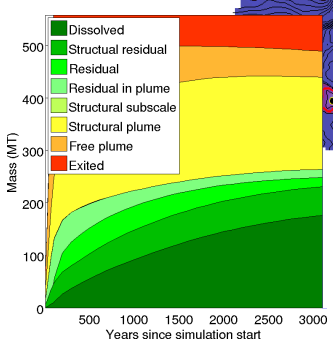


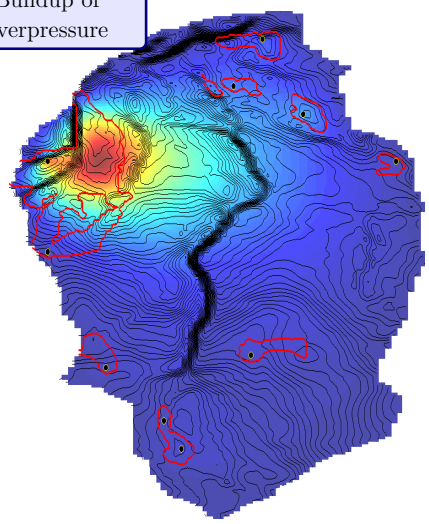
Simple sharp-interface VE simulation





VE simulation with dissolution and subscale trapping





Bulidup of overpressure