Recent developments in the Matlab Reservoir Simulation Toolbox (MRST)

Implicit schemes for compositional flow and hybrid schemes for CO2 storage

Olav Møyner

Department of Mathematics and Cybernetics, SINTEF Digital
Department of Mathematical Sciences, NTNU, Norway

OPM Meeting, October 18, 2017, Bergen

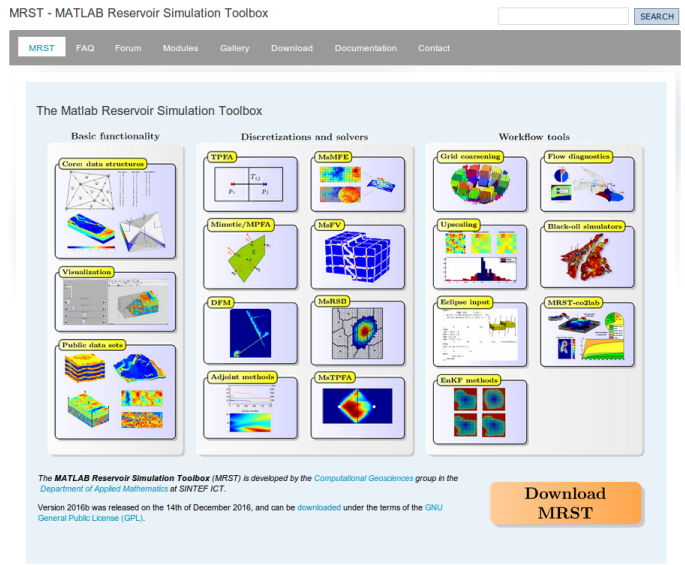
Outline

The MATLAB Reservoir Simulation Toolbox

Open-source toolbox for reservoir modelling, developed by SINTEF Digital and used in most of our research

Wide international user base:

- academic institutions, oil and service companies
- USA, Norway, China, Brazil, United Kingdom, Iran, Germany, Netherlands, France, Canada, ...
- 12 000+ unique downloads since 2013



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

Used in publications:

- 100+ master and PhD theses
- 140+ journal/proceedings papers by authors outside our group

Toolbox for experimental programming

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

Diagnostics/proxies

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

Model reduction

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

Full simulation

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

Two different programming paradigms

incomp – sequential solvers for incompressible flow

- Have been part of MRST since the start
- Uses imperative programming: functions that operate mainly on vectors, (sparse) matrices, structures, and a few cell arrays
- Explicit assembly and linearization of flow equations

AD-OO – (fully) implicit solvers for compressible flow

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

Both families rely on functionality from `mrst-core`

Advanced simulators: motivation

First fully-implicit black-oil implementation: ad-fi (2013)

- very successful in terms of research output
- intended for black-oil with adjoints, but was used as general simulator

However, ad-fi was a victim of own success:

- mixing logic of Newton solver with definition of model equations
- time-stepping and stabilization done per-model
- lots of code duplication and impossible to maintain

Next step: object-orientation

Introduce object-orientation to separate:

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

Only expose needed details and enable more reuse of functionality that has already been developed

Next step: object-orientation

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

- standardized interfaces make Newton solver independent of the specifics of the physical model
- normalized input/output makes it easy to compare and plot results
- switching linear solvers or time-stepping strategy is straightforward

  (Compare `ad-blackoil` and `blackoil-sequential`)
- General sensitivities/gradients through adjoints

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

General setup of simulator

Input deck

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

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

Input parser

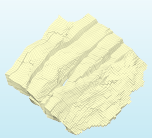
Reads complete simulation decks: grid and petrophysics, fluid and rock properties, region information, well definitions, operating schedule, convergence control, etc

Data
Class
Struct
Function
Input
Contains

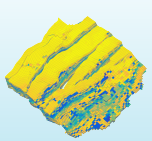
Reservoir model

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

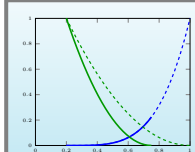
- **Grid**



- **Petrophysics**



- **Fluids**



State

Physical variables inside the reservoir

$\mathtt{p}, \mathtt{s_w}, \mathtt{s_o}, \mathtt{s_g}, \mathtt{c}, \mathtt{r_v}, \mathtt{r_s}$

Well state

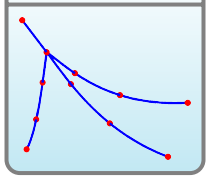
Physical variables inside the wellbore

$\mathtt{q_w^s}, \mathtt{q_o^s}, \mathtt{q_g^s}, \mathtt{q_p^s}, \mathtt{p_{bh}}$

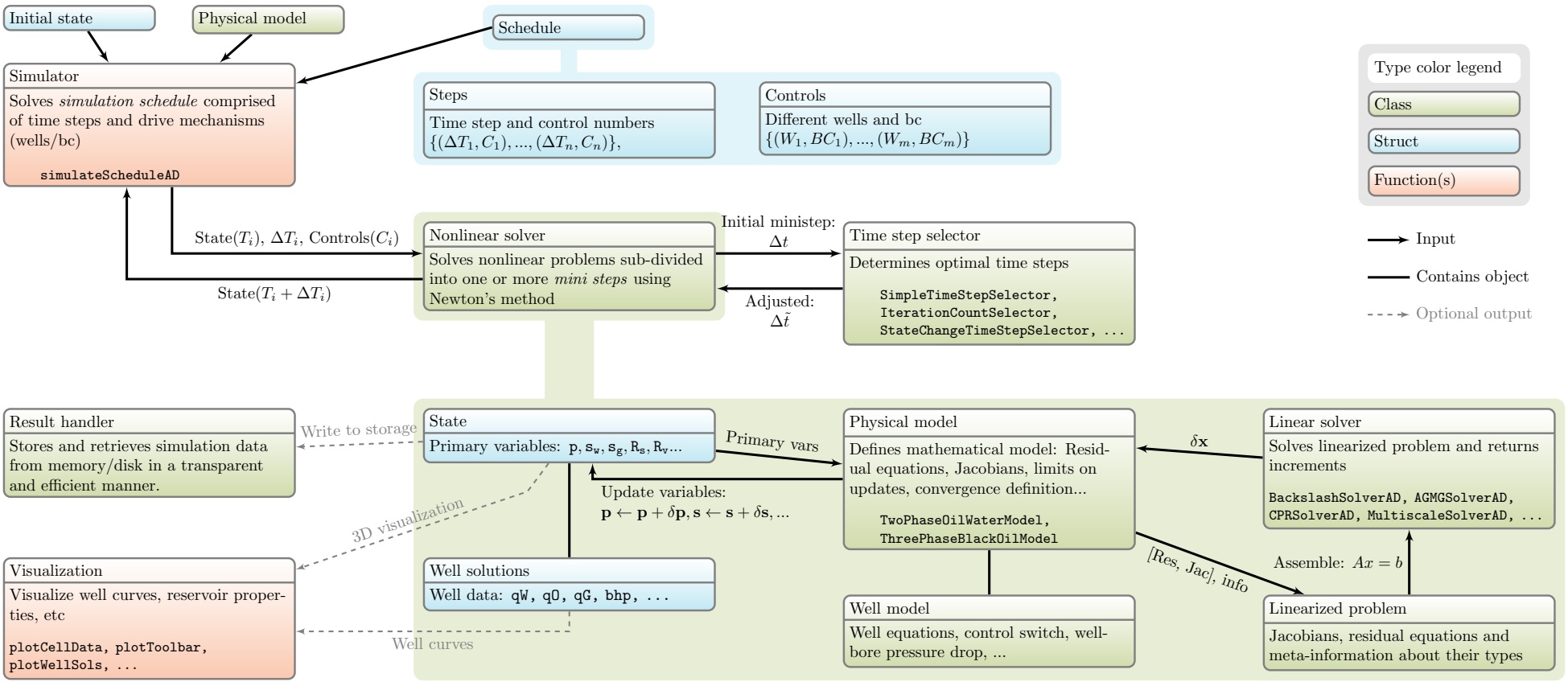
Schedule

Time steps and controls and settings for wells and boundary conditions

- **Wells**

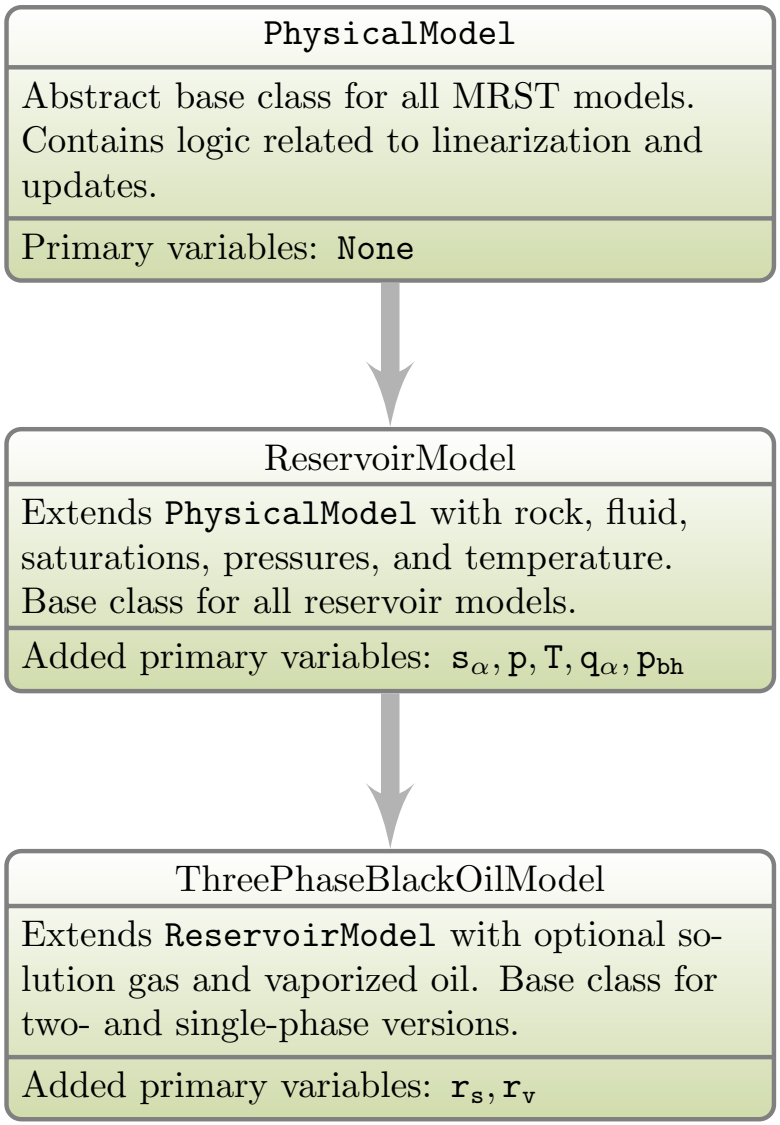


The layout of the AD solvers

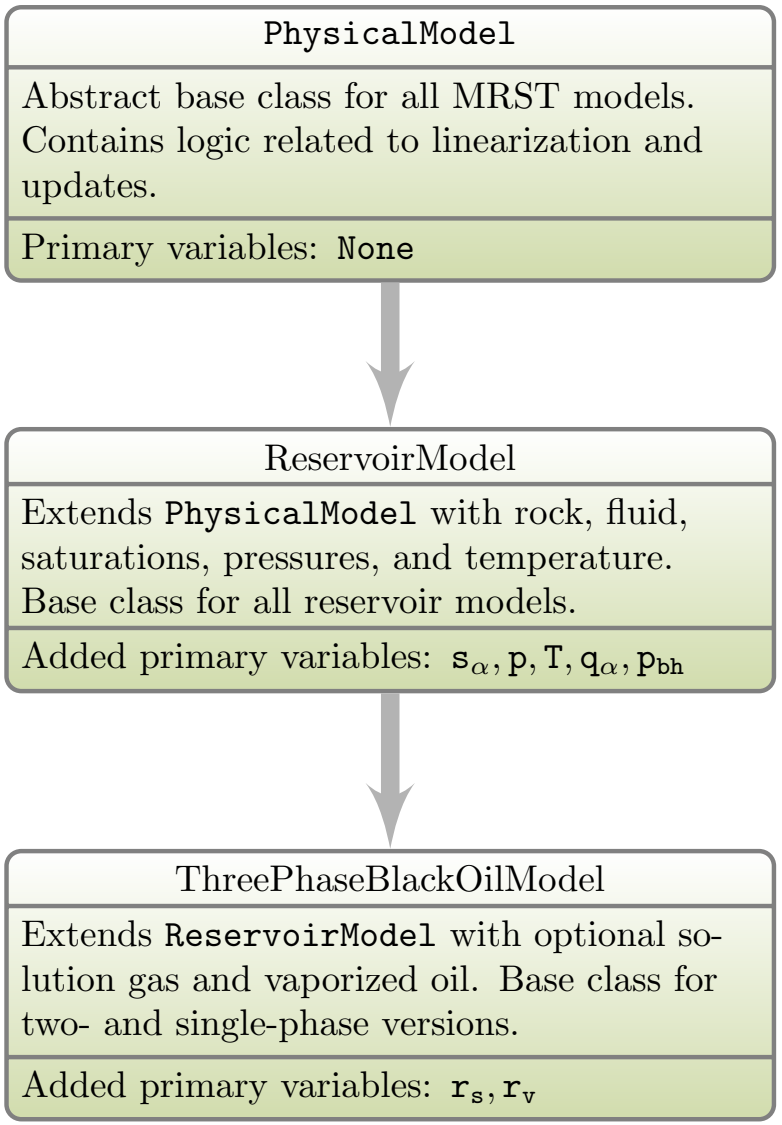


The framework is designed so that you can only work on the components you are interested in: If you want to write a flow solver, you do not need to debug a Newton solver.

Functionality through inheritance



Functionality through inheritance

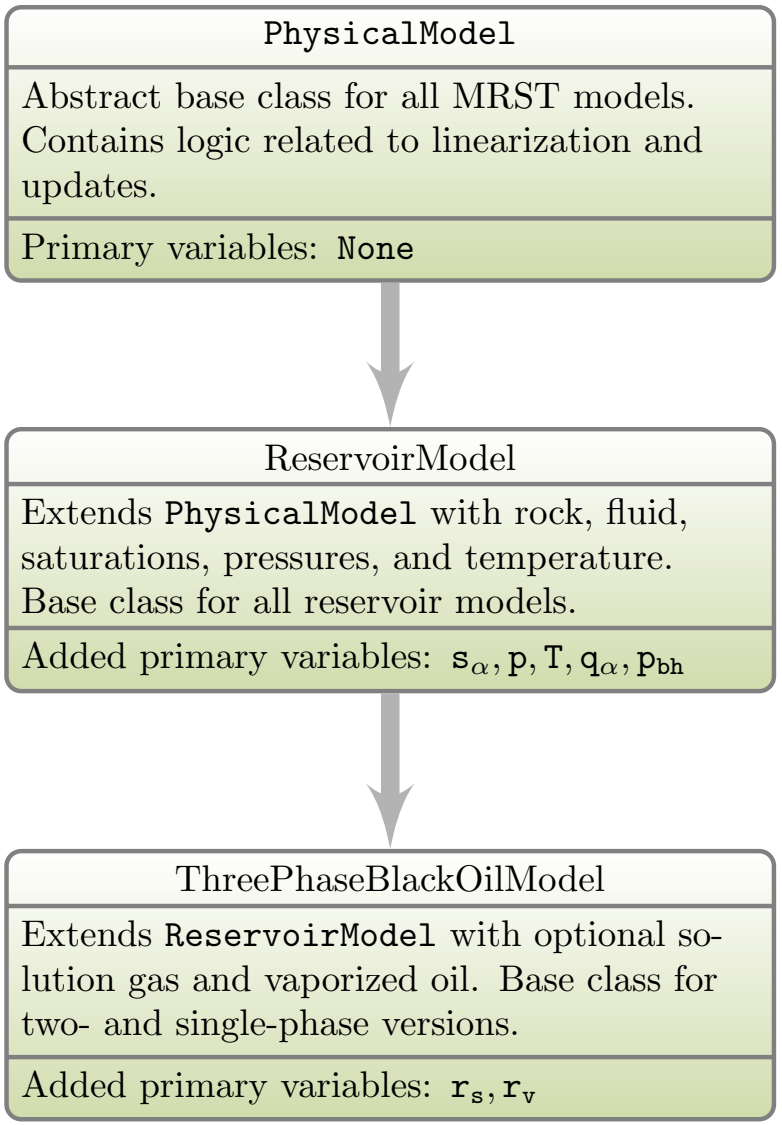


PhysicalModel

Properties:

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

Functionality through inheritance



PhysicalModel

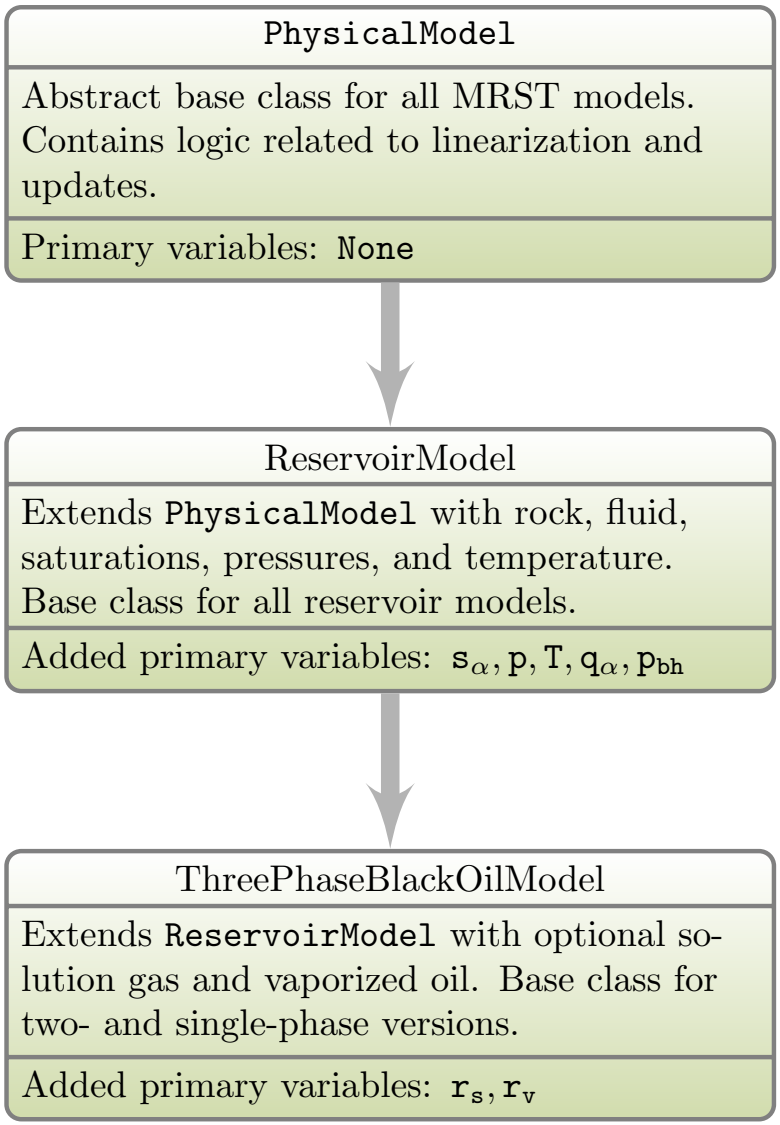
Properties:

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

Quality assurance:

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

Functionality through inheritance



PhysicalModel

Properties:

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

Quality assurance:

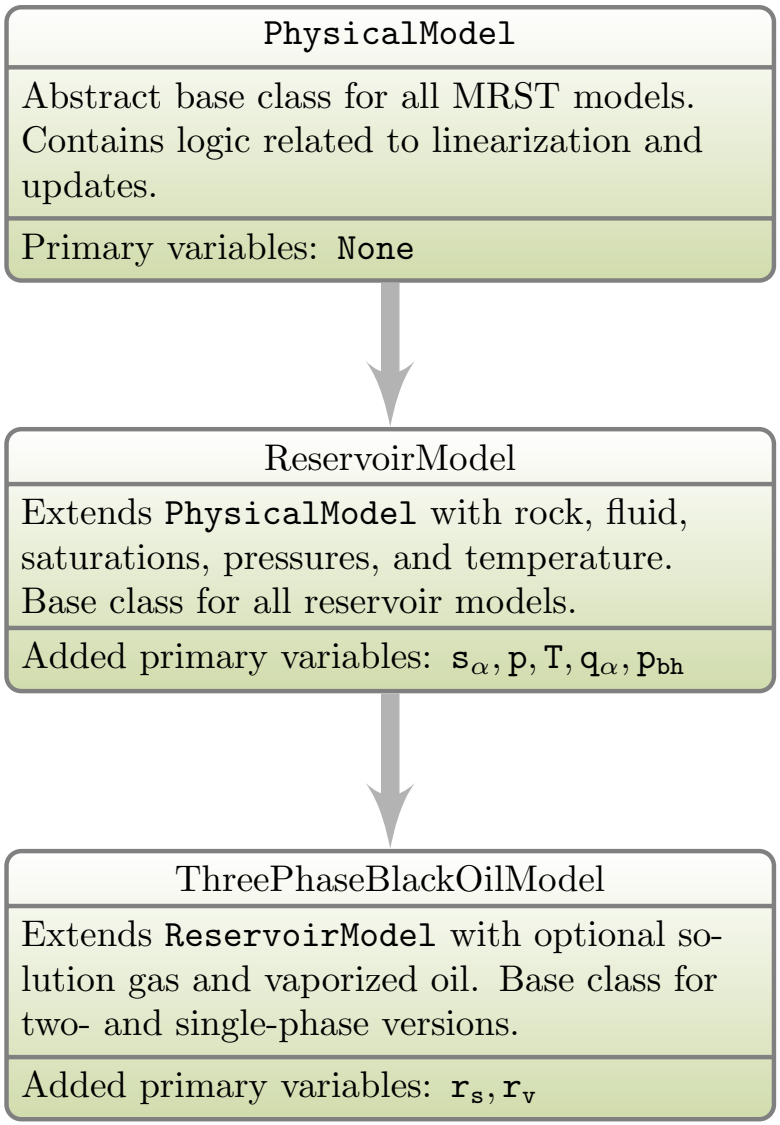
```
state = model.validateState(state)
model = model.validateModel(...)
```

Querying / setting model properties:

```
p     = model.getProp(state, 'pressure')
[p,s] = model.getProps(state, 'pressure', 's')
[f,i] = model.getVariableField(name)
state = model.setProp(model, state, 'pressure', 5)
state = model.incrementProp(state, 'pressure', 1)
state = model.capProperty(state,'saturation', 0, 1)
```

These are examples of syntax for derived classes and will not work on a PhysicalModel, which has no associated variables

Functionality through inheritance

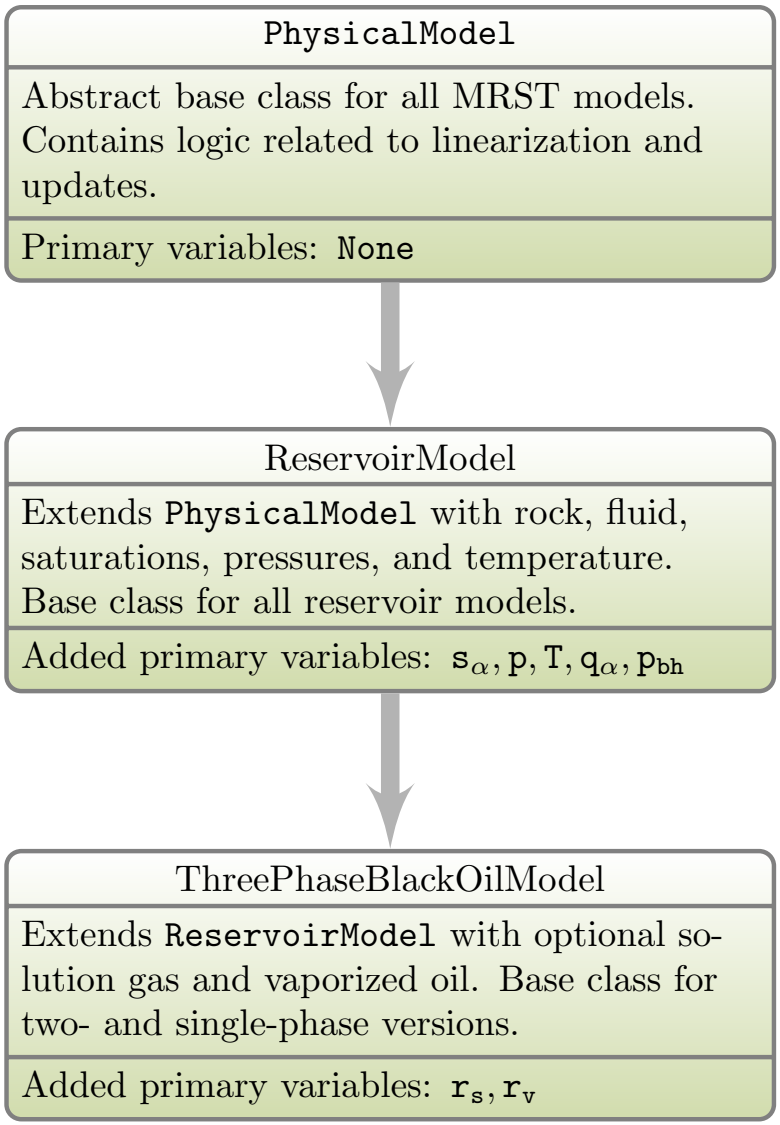


PhysicalModel

Get drive mechanisms:

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

Functionality through inheritance



PhysicalModel

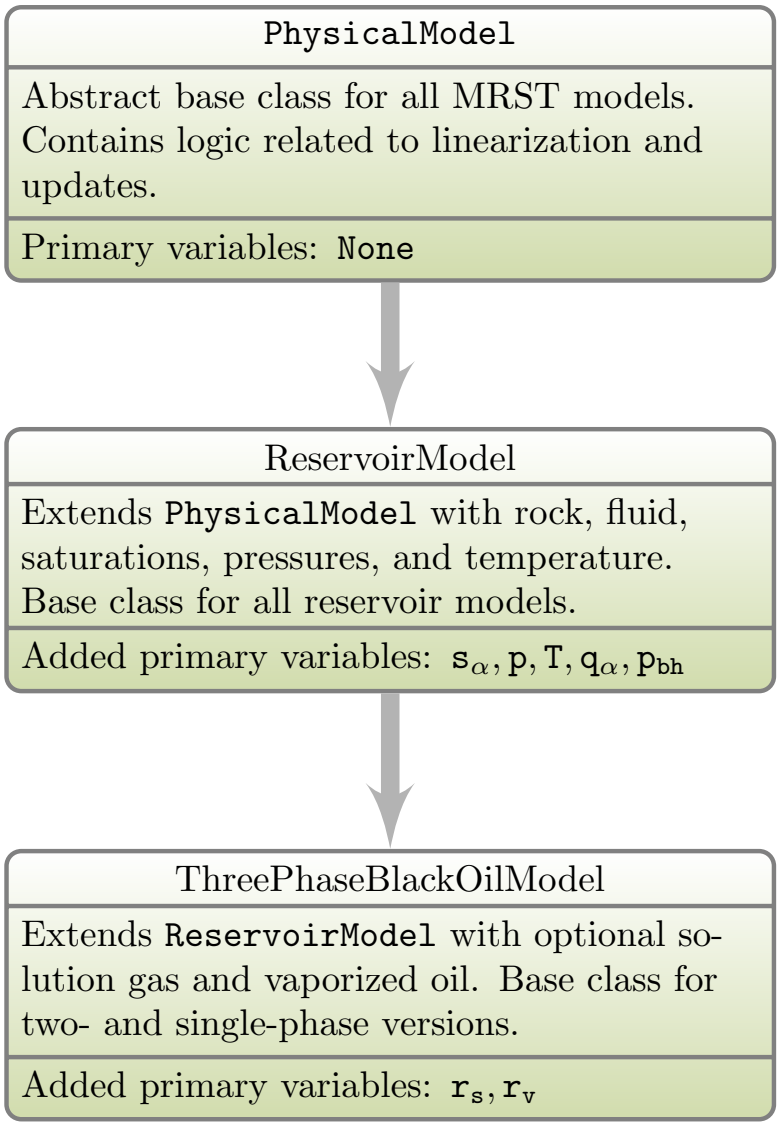
Get drive mechanisms:

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

Linearize and assemble discrete problem:

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

Functionality through inheritance



PhysicalModel

Get drive mechanisms:

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

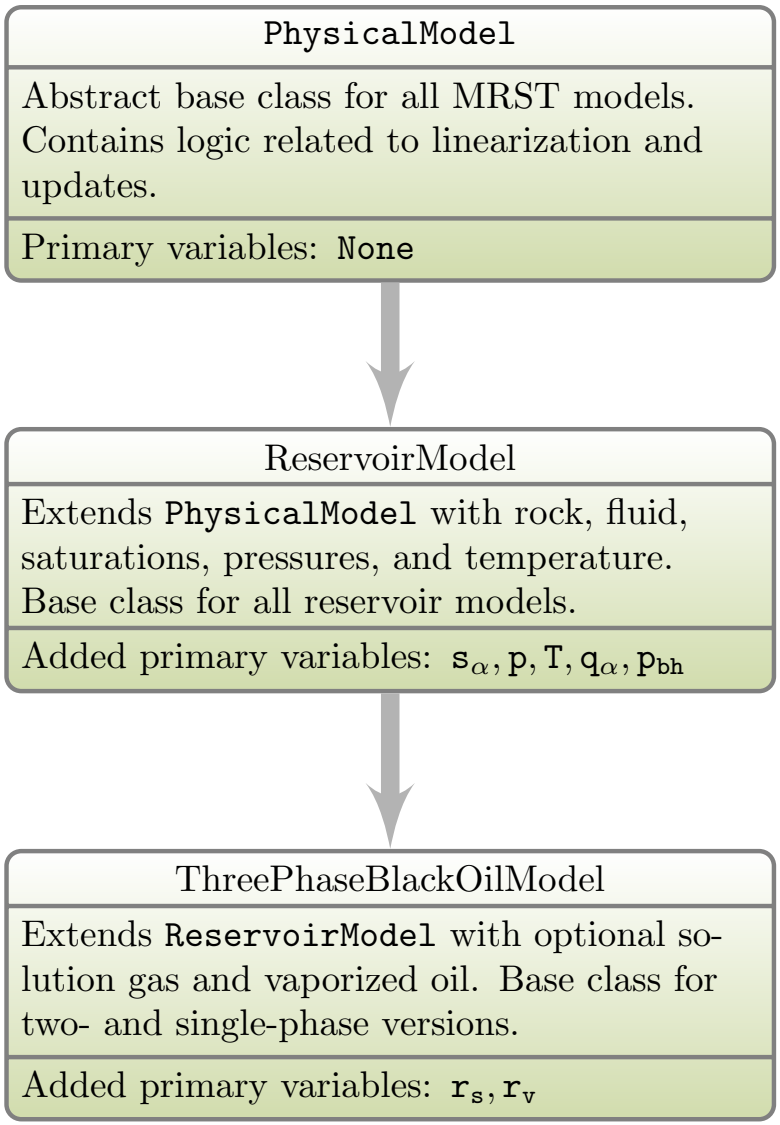
Linearize and assemble discrete problem:

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

Compute a linearized time step:

```
[state, report] = ...
     model.stepFunction(model, state, state0, ..
              dt, drivingForces, linsolve, ...
              nonlinsolve, iteration, varargin)
```

Functionality through inheritance



PhysicalModel

Update state from Newton increment:

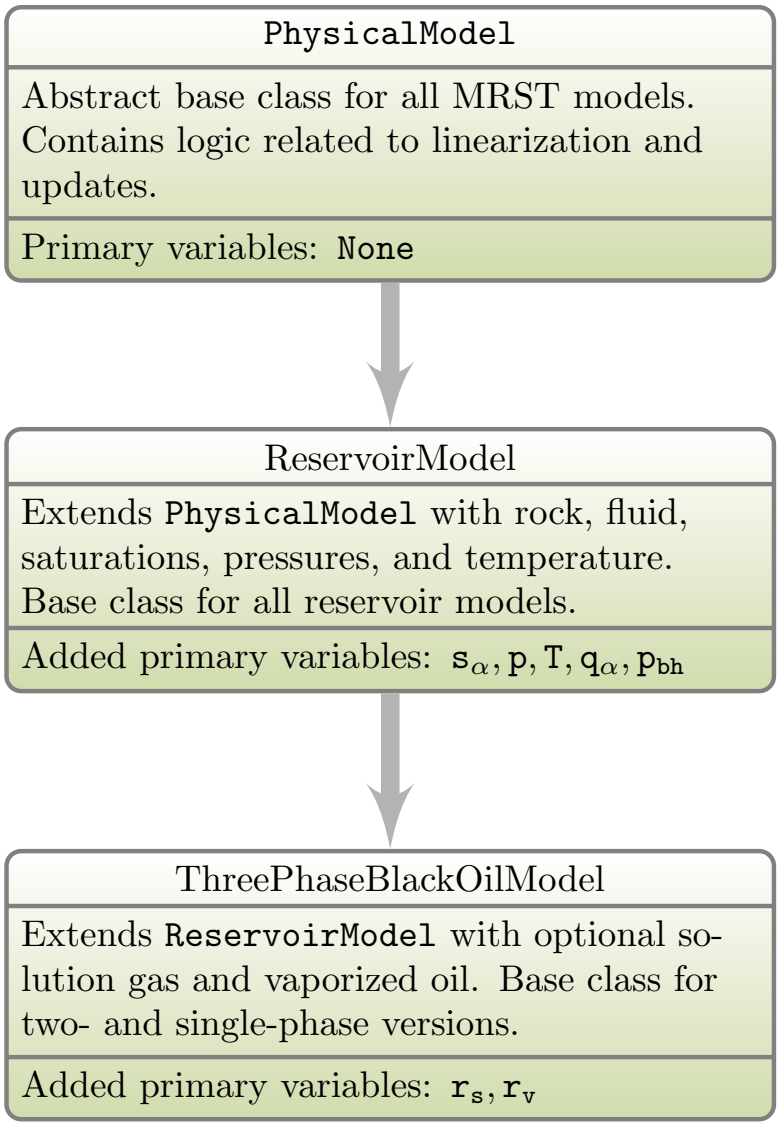
```
[state, report] = model.updateState(state, ...
                                    problem, dx, drivingForces)
```

and other utility functions:

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

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

Functionality through inheritance



ReservoirModel

Properties:

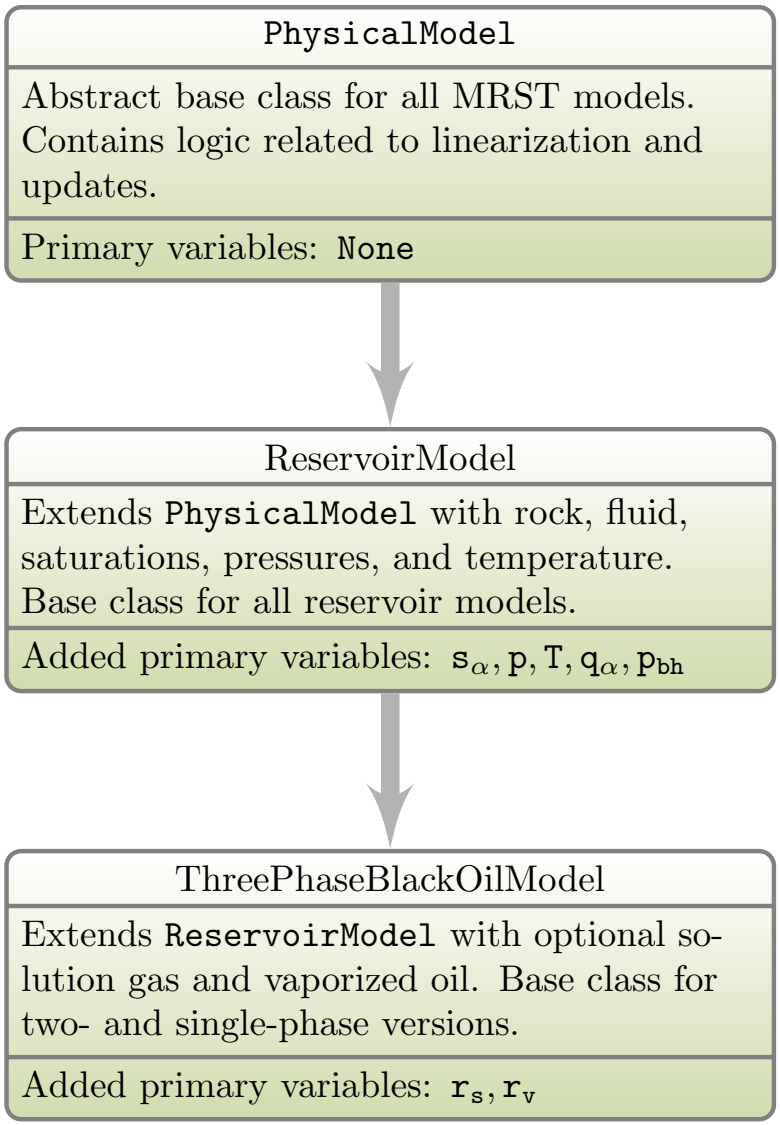
```
% Submodels
fluid, rock, gravity
FacilityModel

% Physical properties
water, gas, oil
saturationVarNames, componentVarNames

% Iterations parameters
dpMaxRel, dpMaxAbs, dsMaxRel, dsMaxAbs
maximumPressure, minimumPressure
useCNVConvergence, toleranceCNV
toleranceMB

% Miscellaneous
:
```

Functionality through inheritance



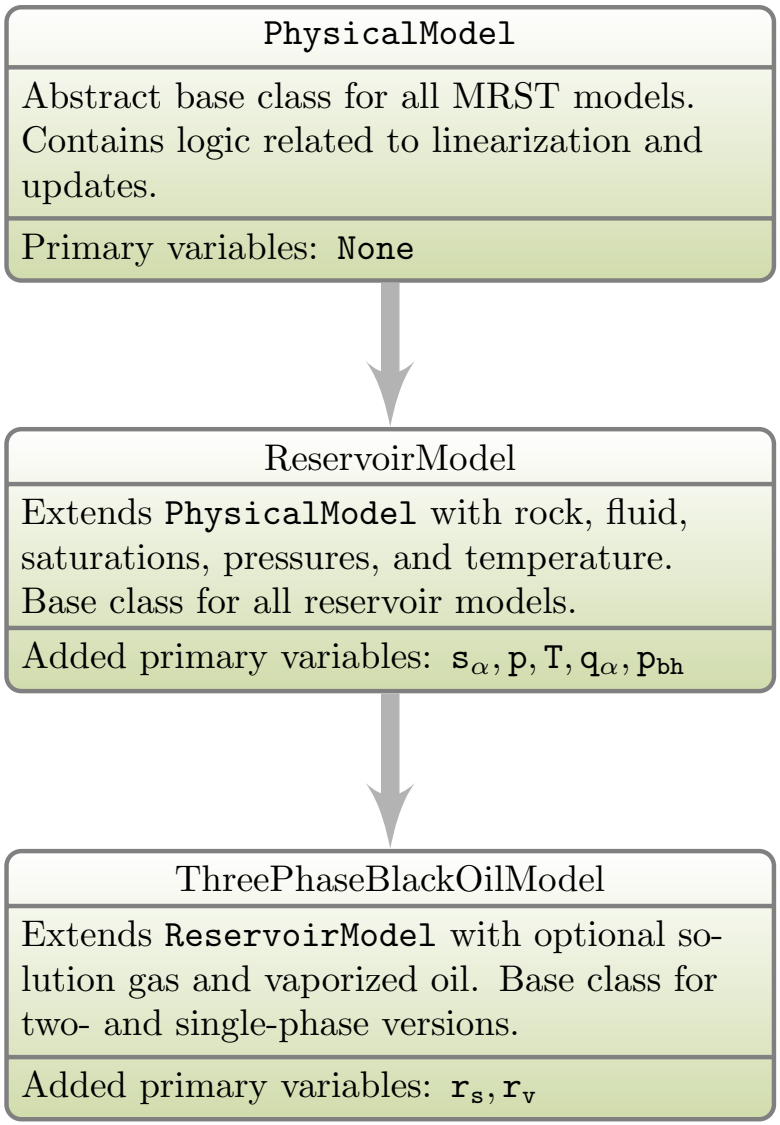
ReservoirModel

Declaration of physical variables:

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

Plus a large number of utility functions to extract, update, and store these physical variables

Functionality through inheritance



ReservoirModel

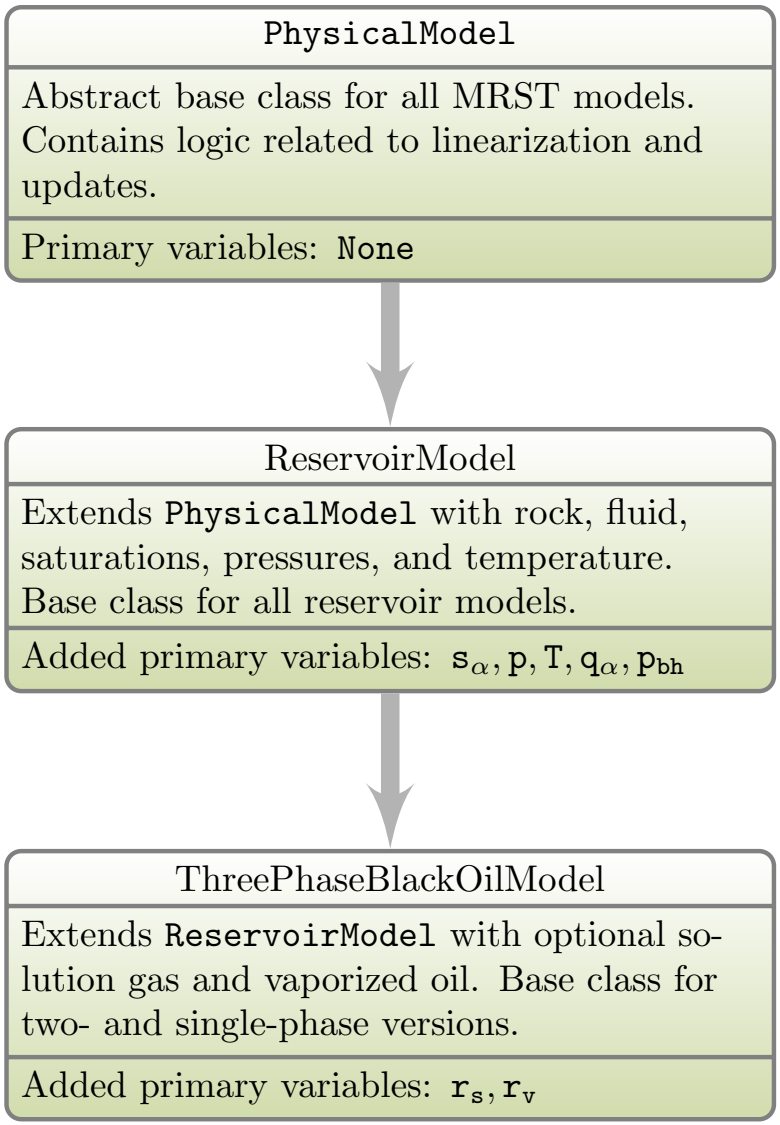
The class declares known drive mechanisms:

```
function forces = getValidDrivingForces(model)
  forces = getValidDrivingForces
           @PhysicalModel(model);
  forces.W   = [];
  forces.bc  = [];
  forces.src = [];
end
```

and define how to evaluate relative permeability, get surface densities, etc.

The class also specifies how to add well equations, source terms, and boundary conditions to the equation system, but does not implement specific flow equations.

Functionality through inheritance

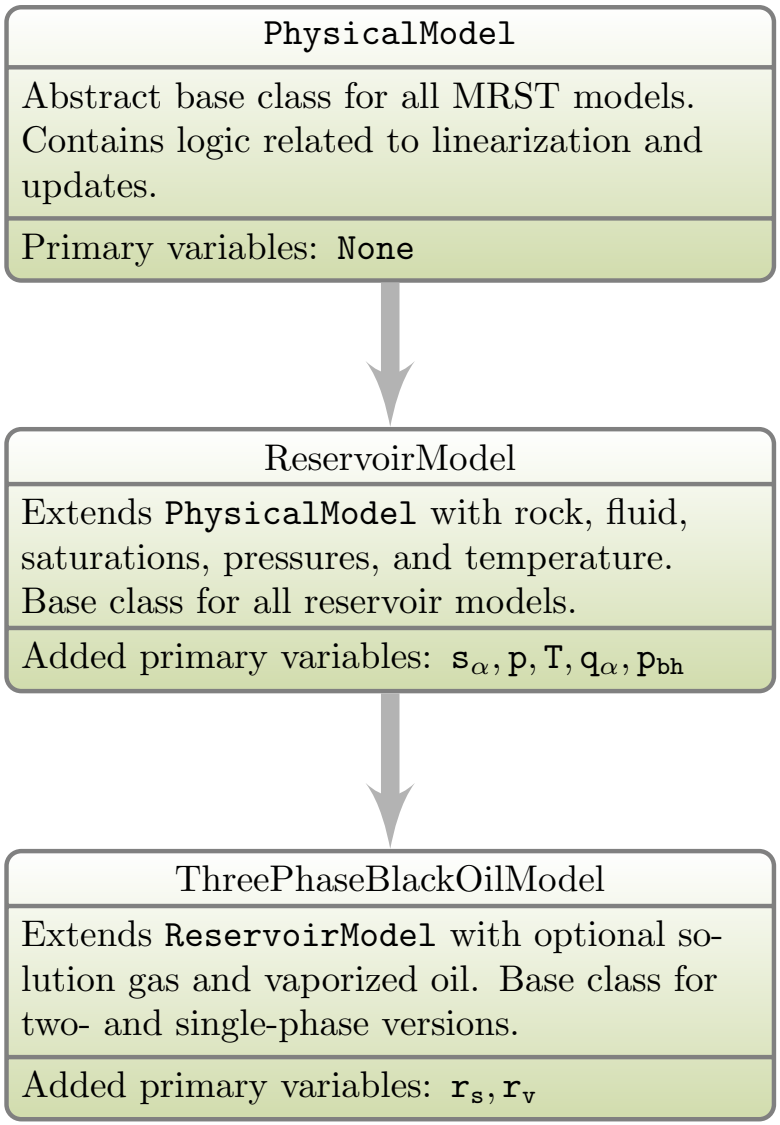


ReservoirModel

Default discretization is a two-point method:

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

Functionality through inheritance



ThreePhaseBlackOilModel

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

Evaluation of residual equations:

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

Details of this function is as given for two-phase case above, but with more features and logic that switches unknowns depending on phases present

Outline

Motivation: Sequential implicit schemes

Separate elliptic/parabolic pressure from hyperbolic transport

- Specialized nonlinear solvers: Multiscale methods for pressure, higher-order transport, reordering methods, trust region solvers, ...
- Efficient: Smaller linearized systems with reduced coupling
- Consensus on scheme for immiscible, weakly compressible

How to treat compositional models in a sequential framework?

Motivation: Sequential implicit schemes

How to treat compositional models? What do we want?

- No time-step restrictions in transport due to e.g. small cells
- Robustness for different flow regimes – avoid oscillations, unphysical values
- Exact mass conservation without outer loop
- Agreement with fully-implicit discretization
- Symmetric: No preferential treatment for specific components!

This work: Fixed-volume formulation. Joint work with Hamdi Tchelepi

Previous works: Acs et al (1985), Watts (1986), Trangenstein & Bell (1989), Coats (1995), Haukås et al (2006), Hajibeygi & Tchelepi (2014), Møyner & Tchelepi(2017), Moncorgé et al (2017)

Governing equations for isothermal flow

- Conservation of each component $i \in \{1, ..., N\}$,

$$\frac{\partial}{\partial t}(\phi \left[\rho_L S_L X_i + \rho_V S_V Y_i\right]) + \nabla \cdot (\rho_L X_i \vec{v}_L + \rho_V Y_i \vec{v}_V) = q_i,$$

  with natural variables: $p, S_L, S_V, x_1, ..., x_N, y_1, , ..., y_N$.

- Fugacity balance for cells with both liquid and vapor

$$f_i^L(p, T, x_1, ..., x_N) = f_i^V(p, T, y_1, ..., y_N).$$

- Sum of fractions close the system,

$$\sum_{i=1}^{N} x_i = 1, \sum_{i=1}^{N} y_i = 1, S_V + S_L = 1.$$

Fluxes are given by multiphase Darcy's law:

$$\vec{v}_\alpha = -K\lambda_\alpha(\nabla p_\alpha - \rho_\alpha g \Delta z)$$

Governing equations for isothermal flow

- Conservation of each component $i \in \{1, ..., N\}$,

$$\frac{\partial}{\partial t}(\phi[\rho_L S_L X_i + \rho_V S_V Y_i]) + \nabla \cdot (\rho_L X_i \vec{v}_L + \rho_V Y_i \vec{v}_V) = q_i,$$

with natural variables: $p, S_L, S_V, x_1, ..., x_N, y_1, , ..., y_N$.

- Fugacity balance for cells with both liquid and vapor

- Sum

Note: $\rho_\alpha, S_\alpha, X_i, Y_i, \lambda_\alpha, ...$ depend strongly on both hyperbolic overall mole fractions z_i and parabolic pressure p, regardless of choice of primary variables.

Fluxes are given by multiphase Darcy's law:

$$\vec{v}_\alpha = -K\lambda_\alpha(\nabla p_\alpha - \rho_\alpha g \Delta z)$$

Governing equations for isothermal flow

Compositional model in the Matlab Reservoir Simulation Toolbox (MRST) has typical choices for compositional simulation of hydrocarbons,

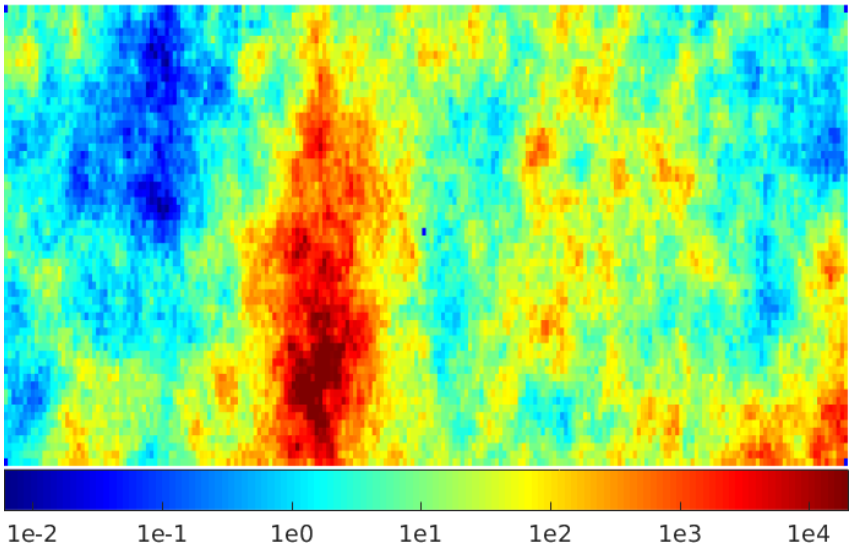
- Densities and phase behavior predicted by equation-of-state
- Generalized cubic equation-of-state: Martin's equation
- Lohrenz-Bray-Clark viscosity correlations
- Schur-complement used to obtain N by N system for variable set α,

$$-J\Delta x = \begin{bmatrix} B & C \\ D & E \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} f \\ h \end{bmatrix} \rightarrow A\alpha = (B - CE^{-1}D)\alpha = f - CE^{-1}h = b.$$

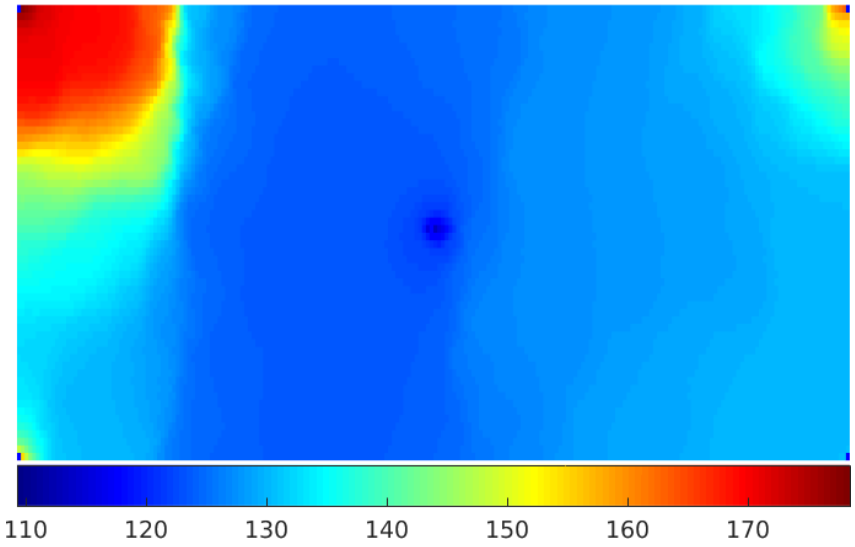
Remark: E is easily invertible, as fugacity is local to each cell

- Support for both natural-variables and overall composition

Example 3: Multiscale SPE10



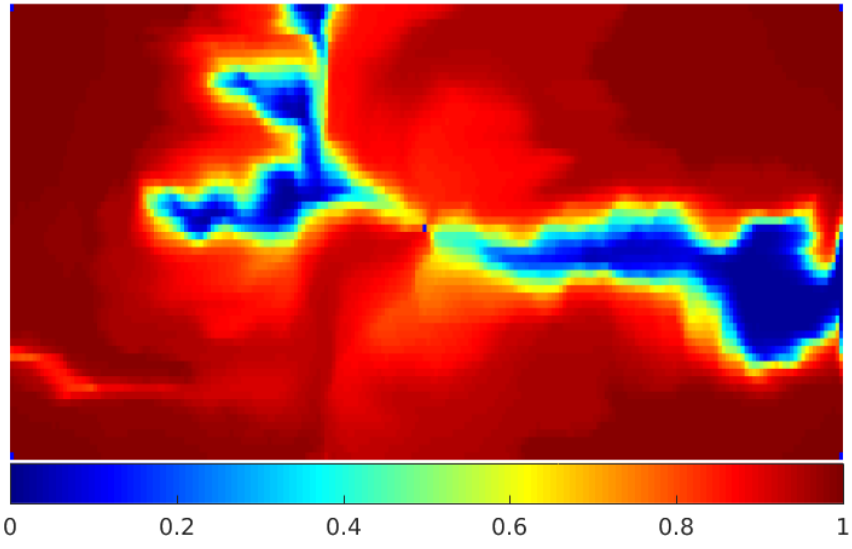
Permeability [mD]



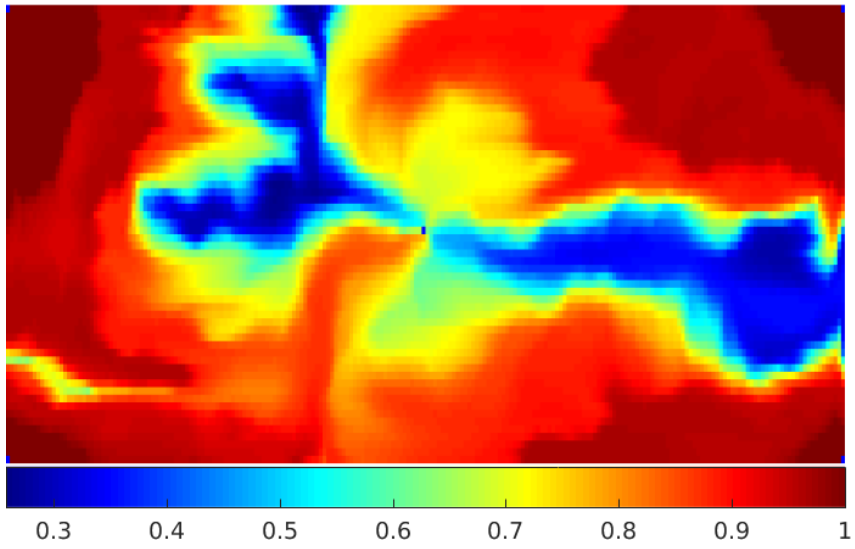
Pressure [bar]

- Inject approximately 1 pore-volume CO_2 over 2000 days
- Layer taken from Tarbert formation, SPE 10, model 2
- Multiscale solver with MsRSB basis functions and 10^{-3} tolerance
- 6 pseudocomponent fluid from Mallison et al

Example 3: Multiscale SPE10



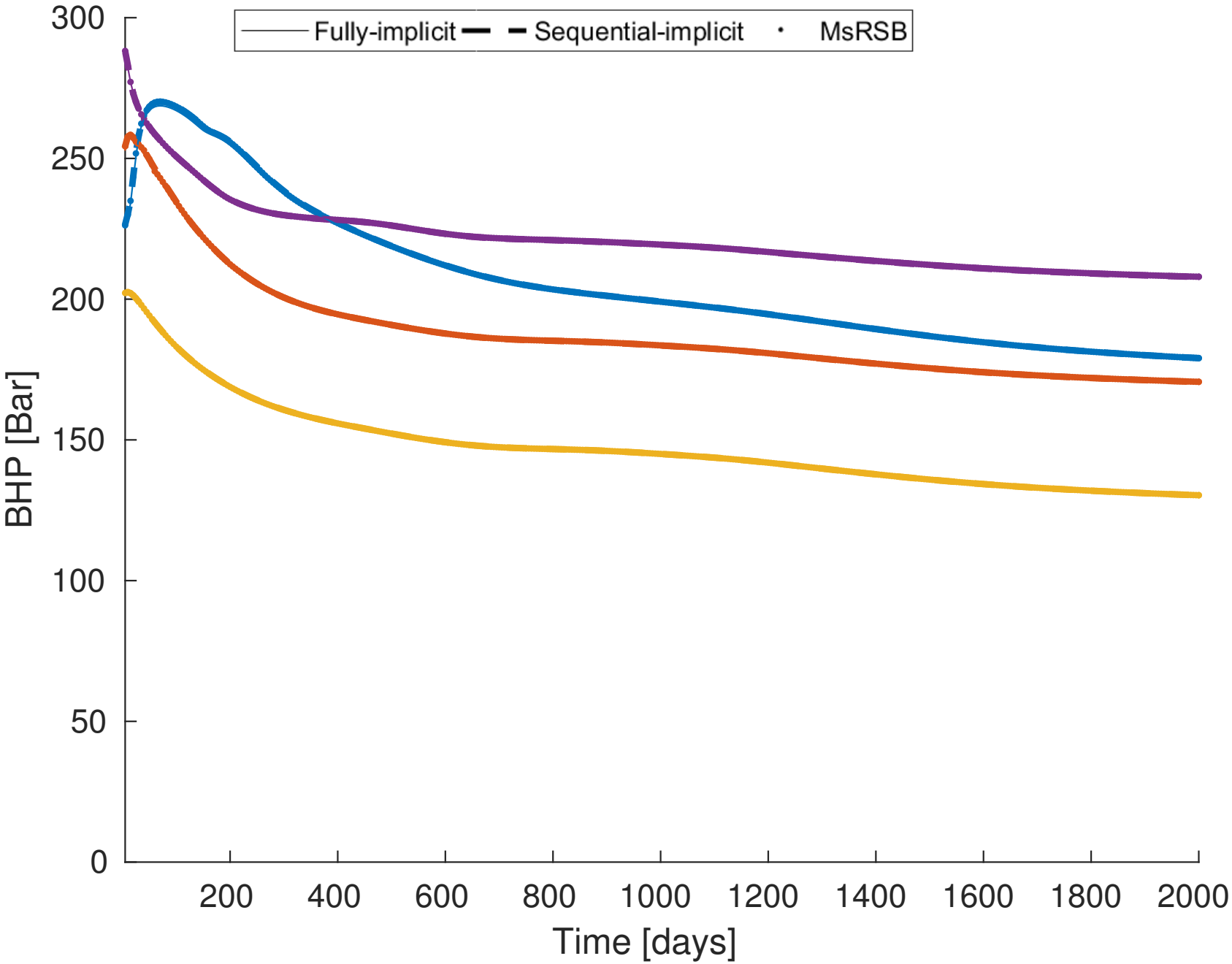
CO_2 mole fraction



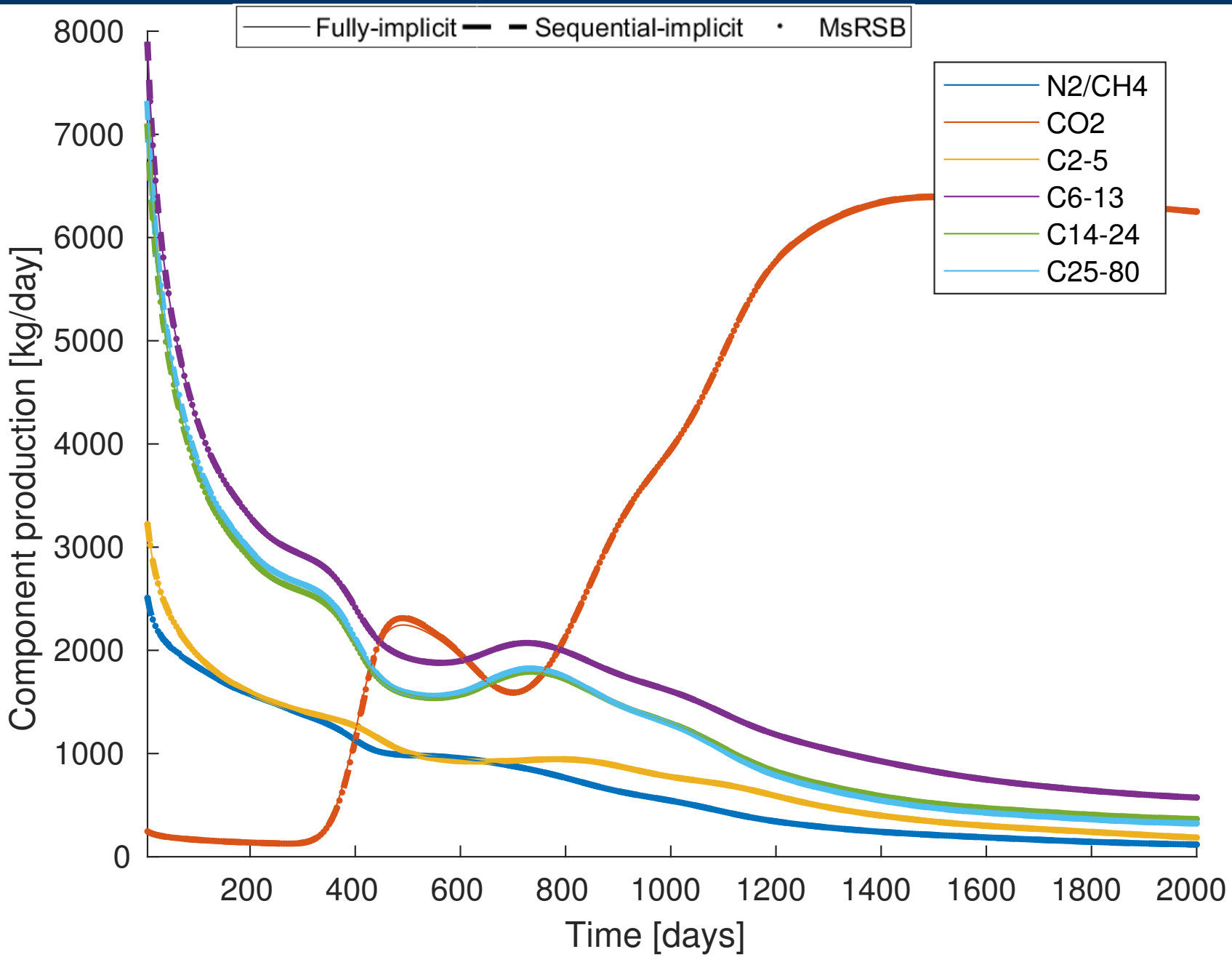
Gas saturation

- Inject approximately 1 pore-volume CO_2 over 2000 days
- Layer taken from Tarbert formation, SPE 10, model 2
- Multiscale solver with MsRSB basis functions and 10^{-3} tolerance
- 6 pseudocomponent fluid from Mallison et al

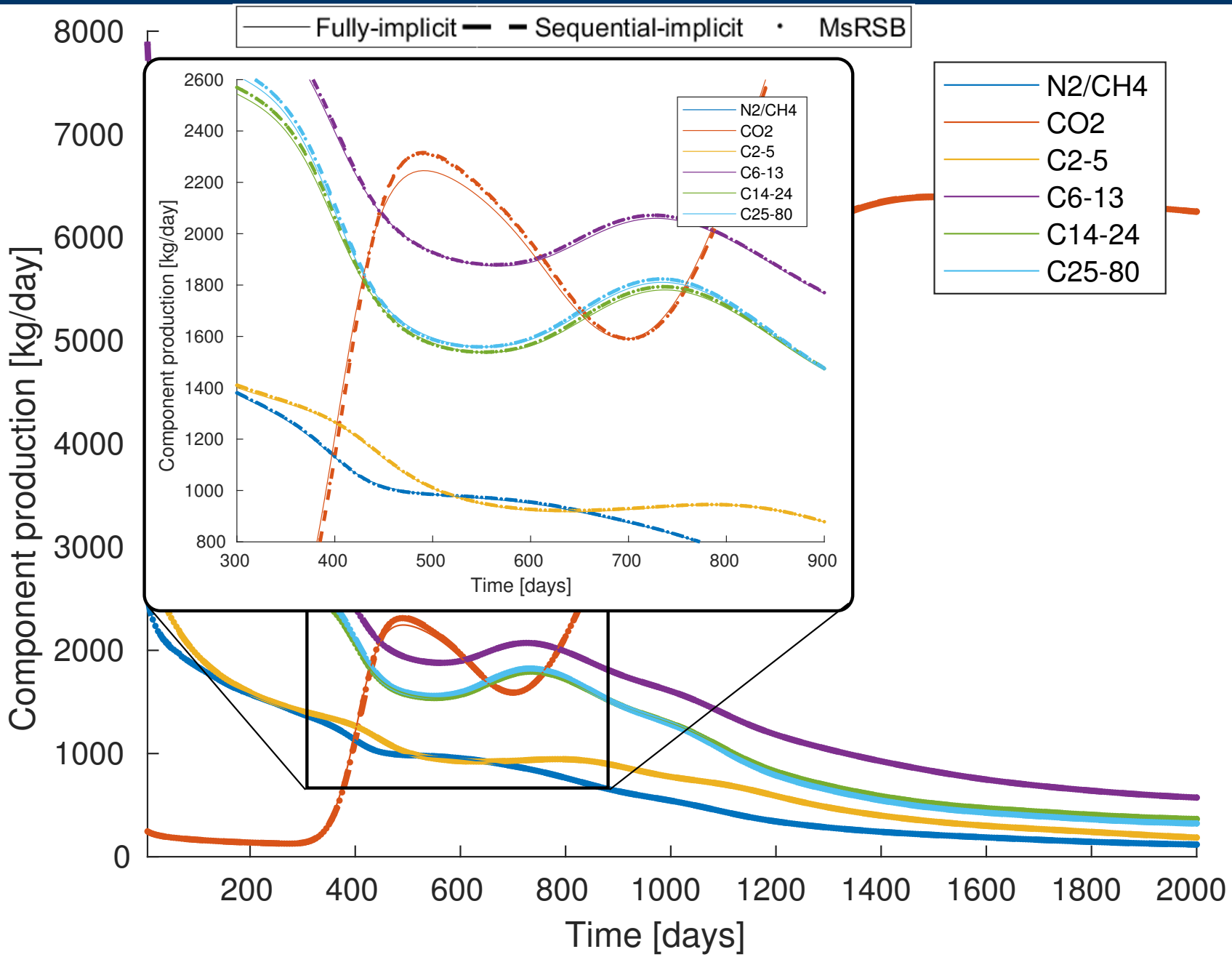
Example: Multiscale SPE10



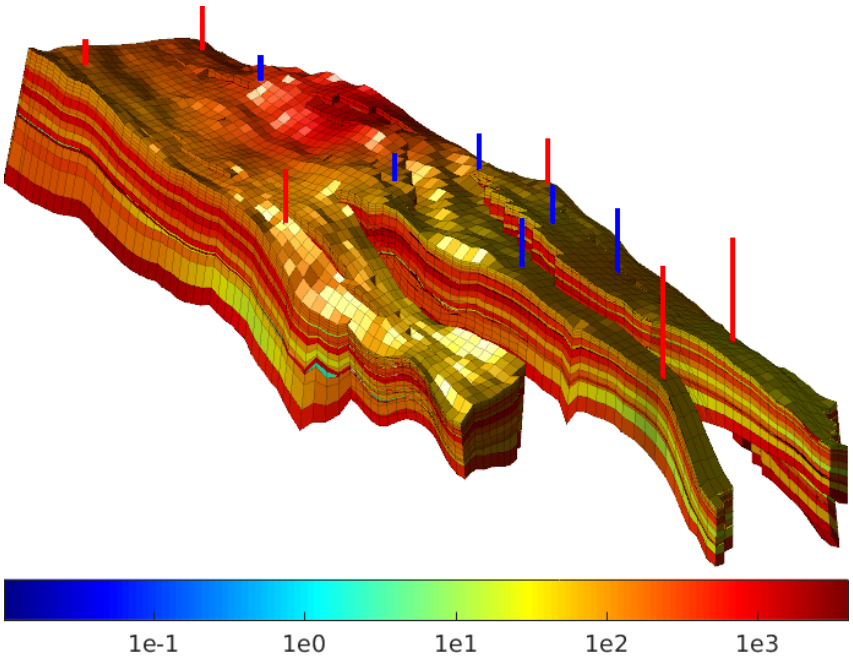
Example: Multiscale SPE10



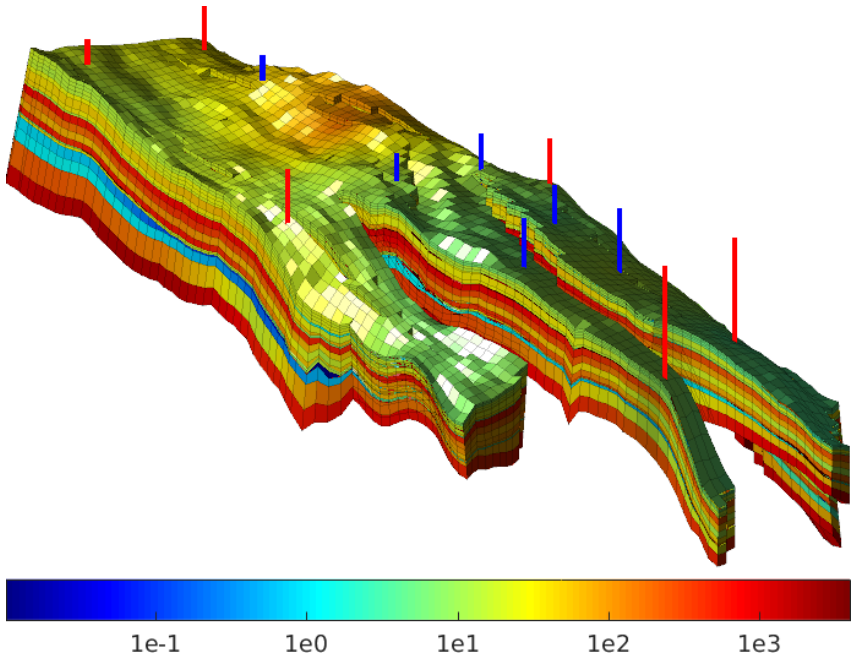
Example: Multiscale SPE10



Example: Multiscale Norne



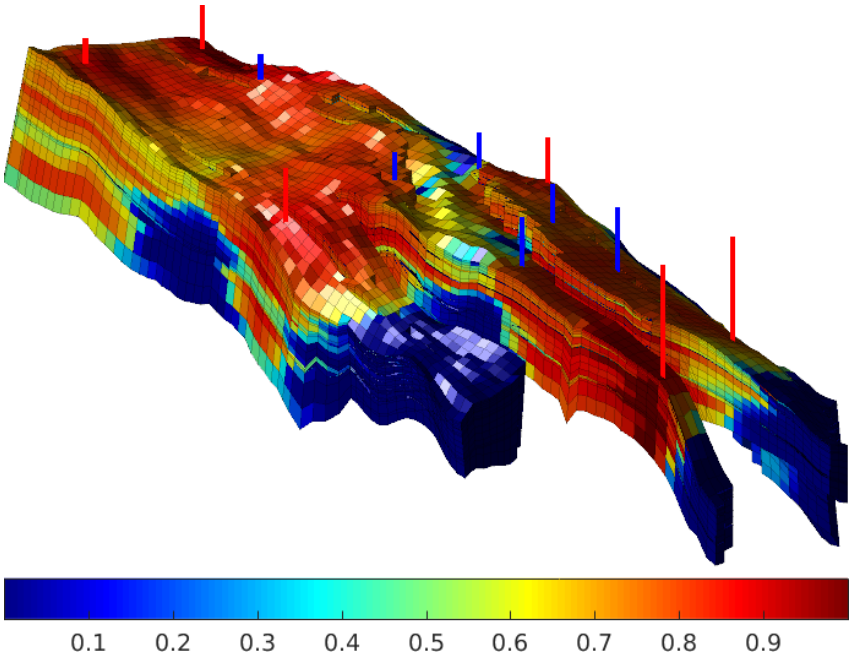
Vertical permeability [mD]



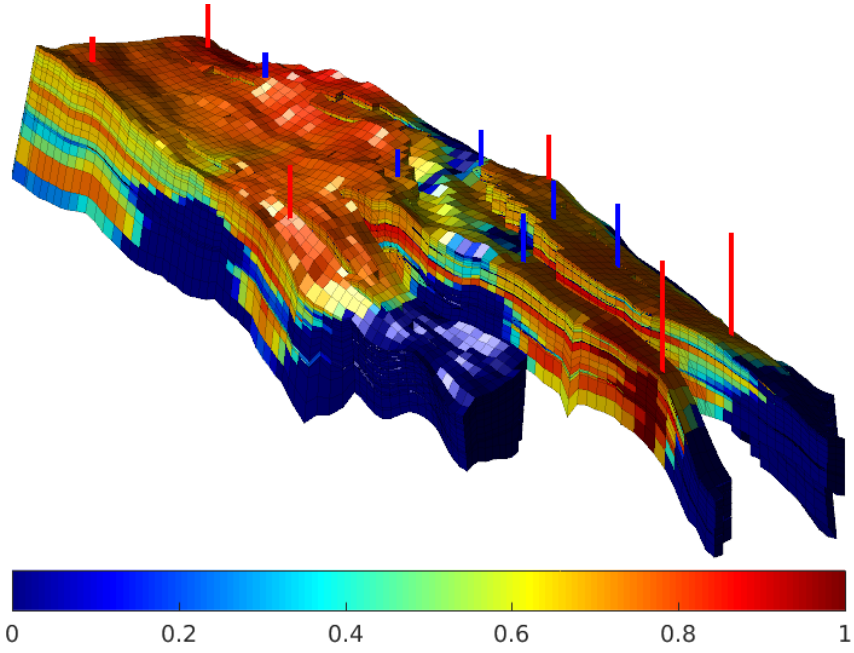
Pressure [bar]

- Nitrogen injection over 2500 days, 4 components
- Faulted, anisotropic model
- Significant gravity effect with wells perforated in five layers
- Multiscale solver with MsRSB basis functions and 10^{-3} tolerance
- Maximum CFL: 160

Example: Multiscale Norne



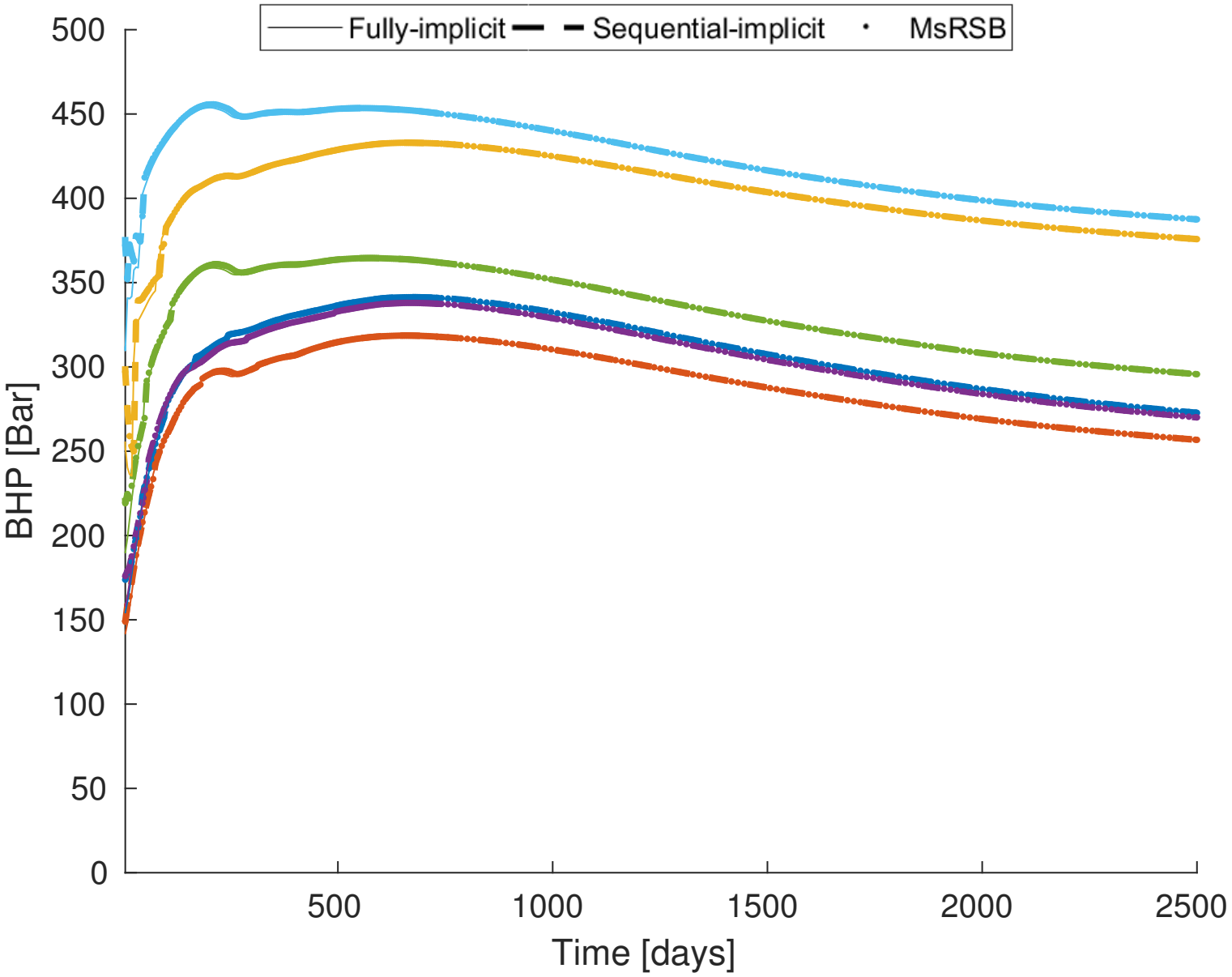
N_2 mole fraction



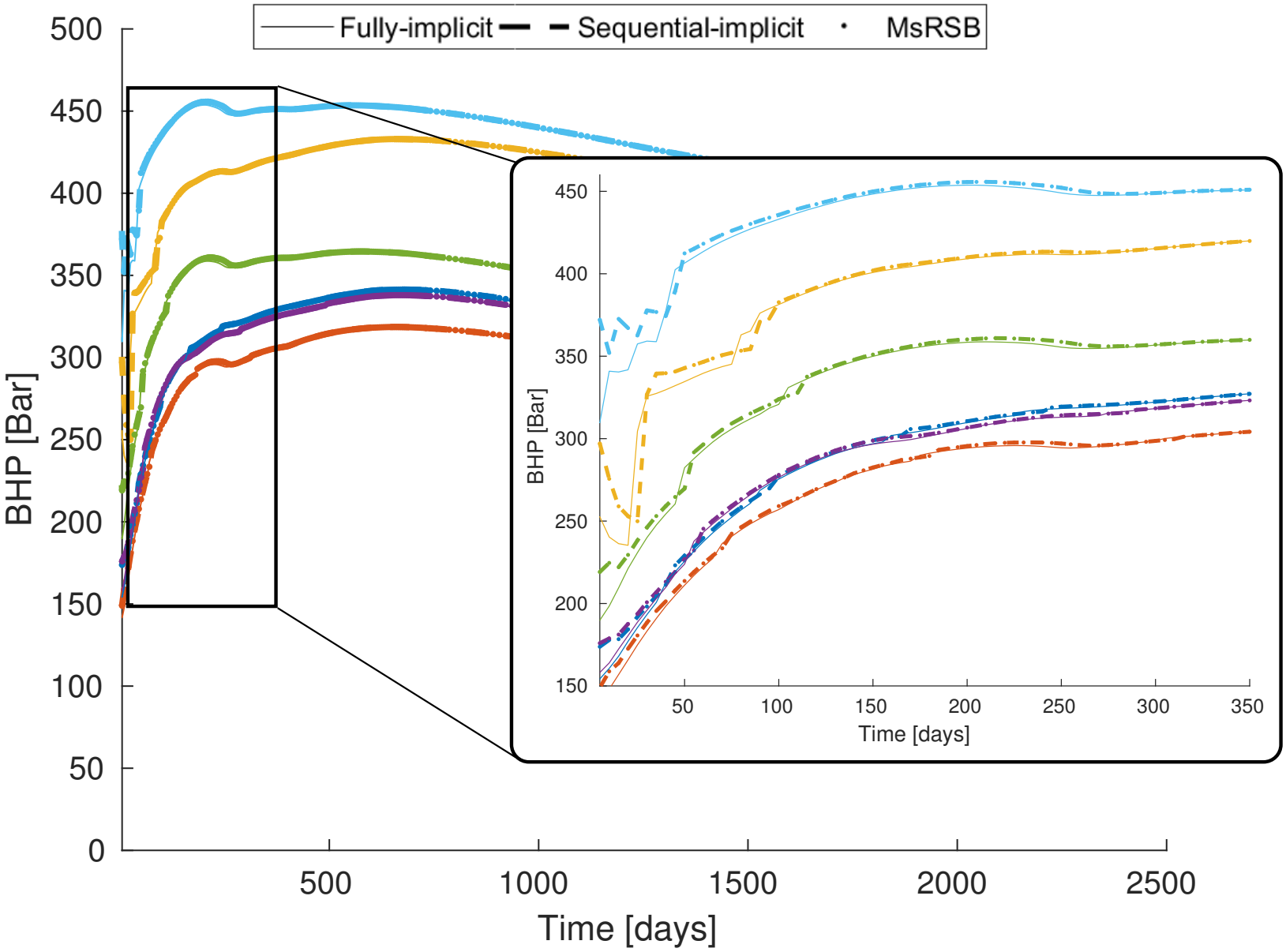
Gas saturation

- Nitrogen injection over 2500 days, 4 components
- Faulted, anisotropic model
- Significant gravity effect with wells perforated in five layers
- Multiscale solver with MsRSB basis functions and 10^{-3} tolerance
- Maximum CFL: 160

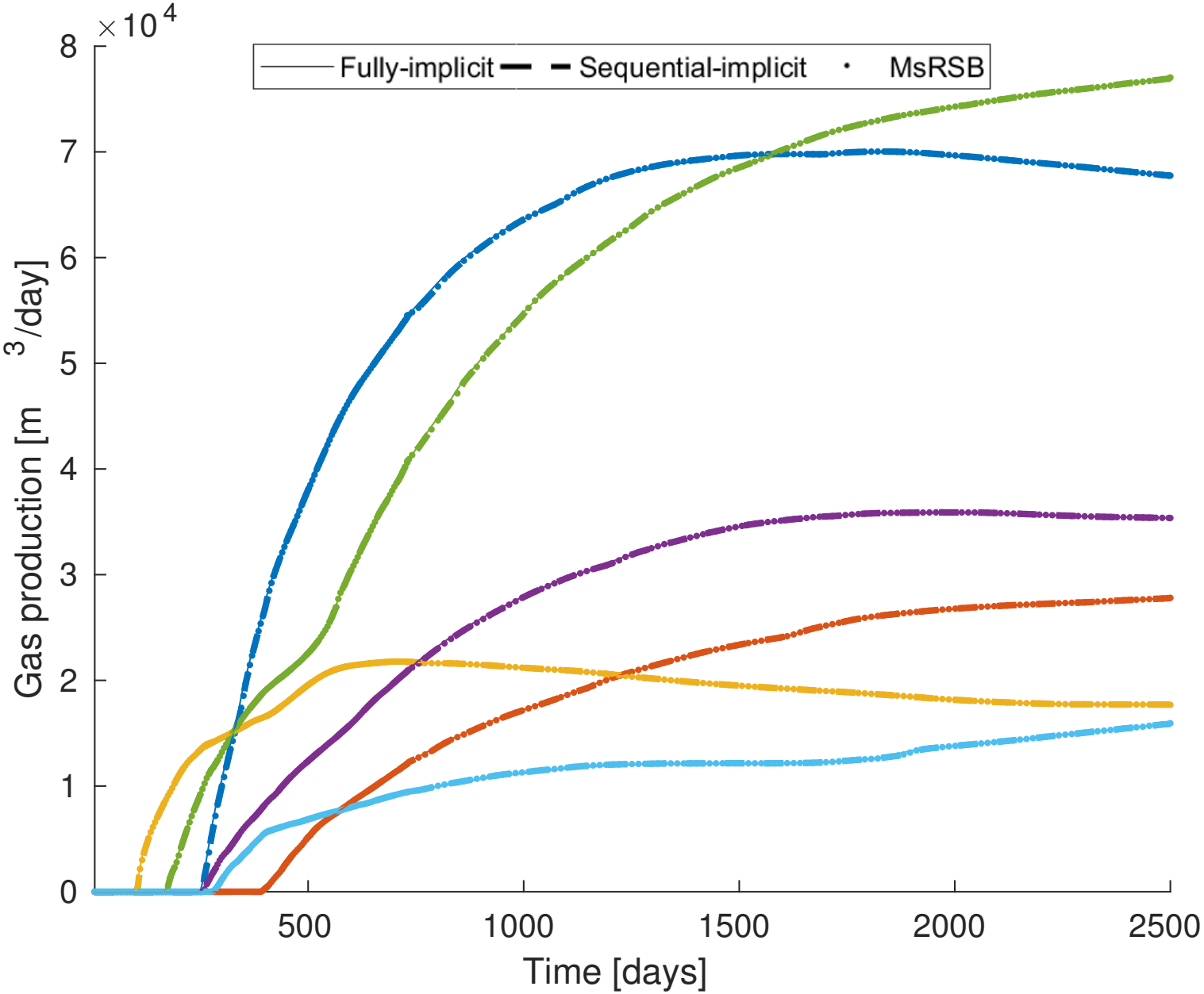
Example: Multiscale Norne



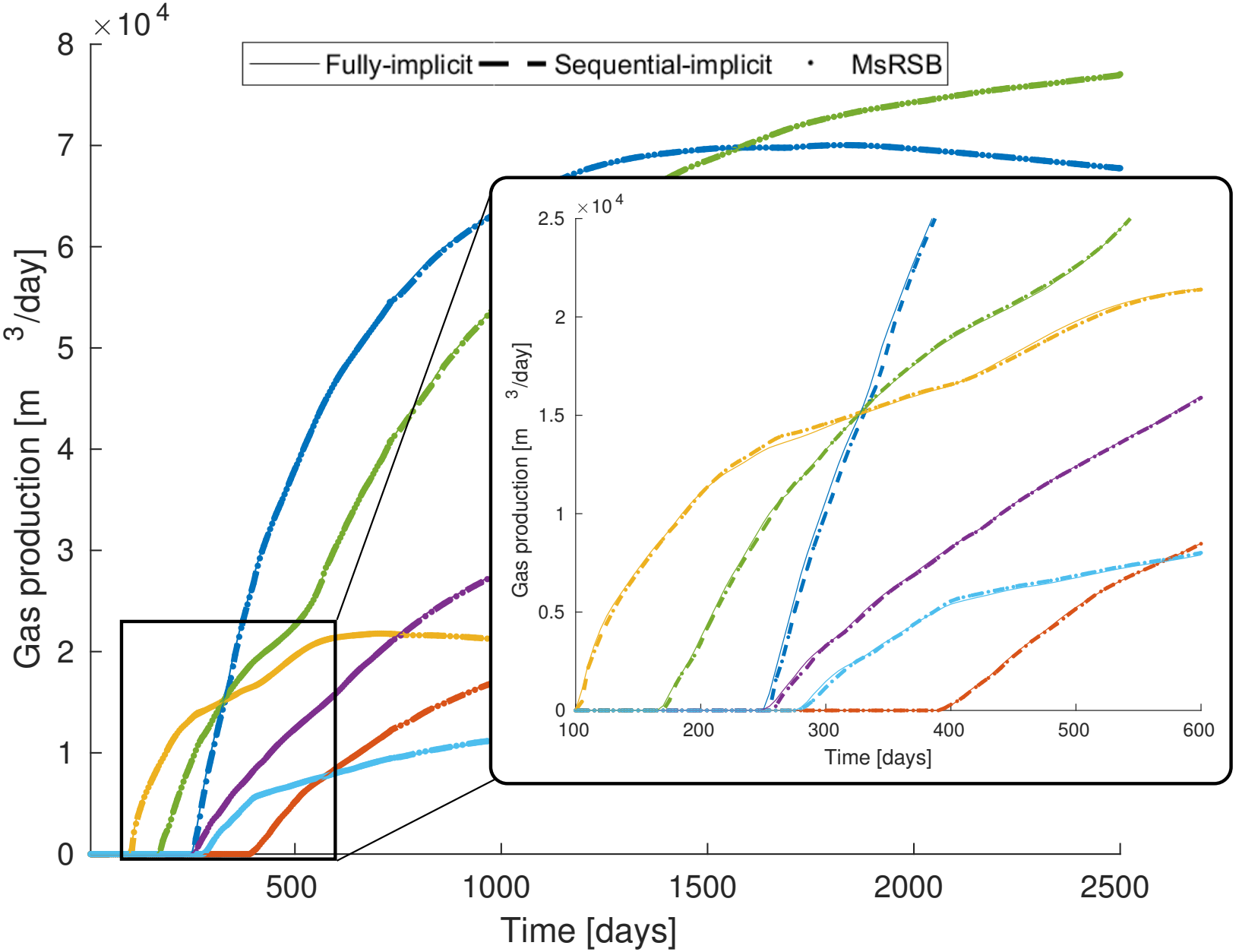
Example: Multiscale Norne



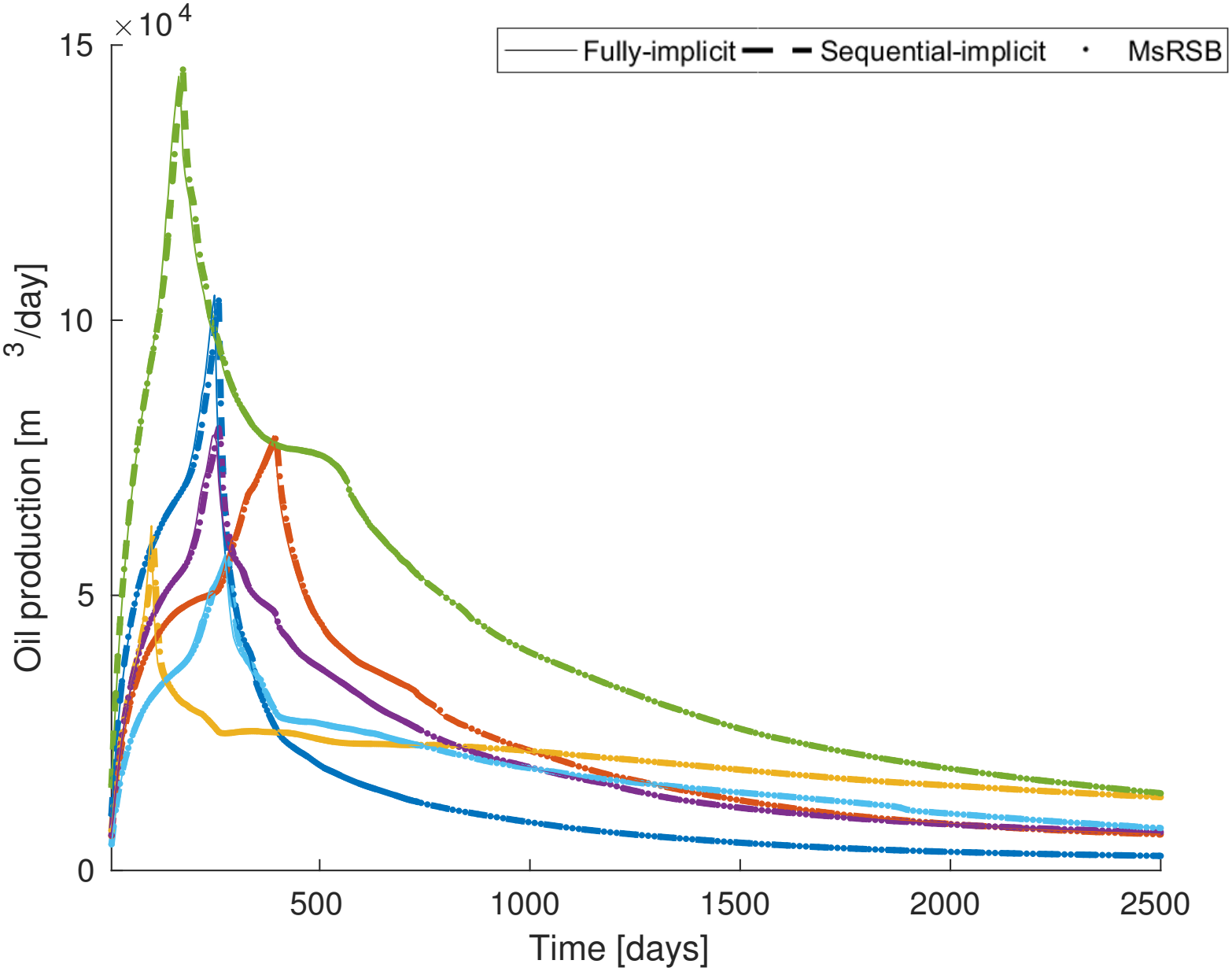
Example: Multiscale Norne



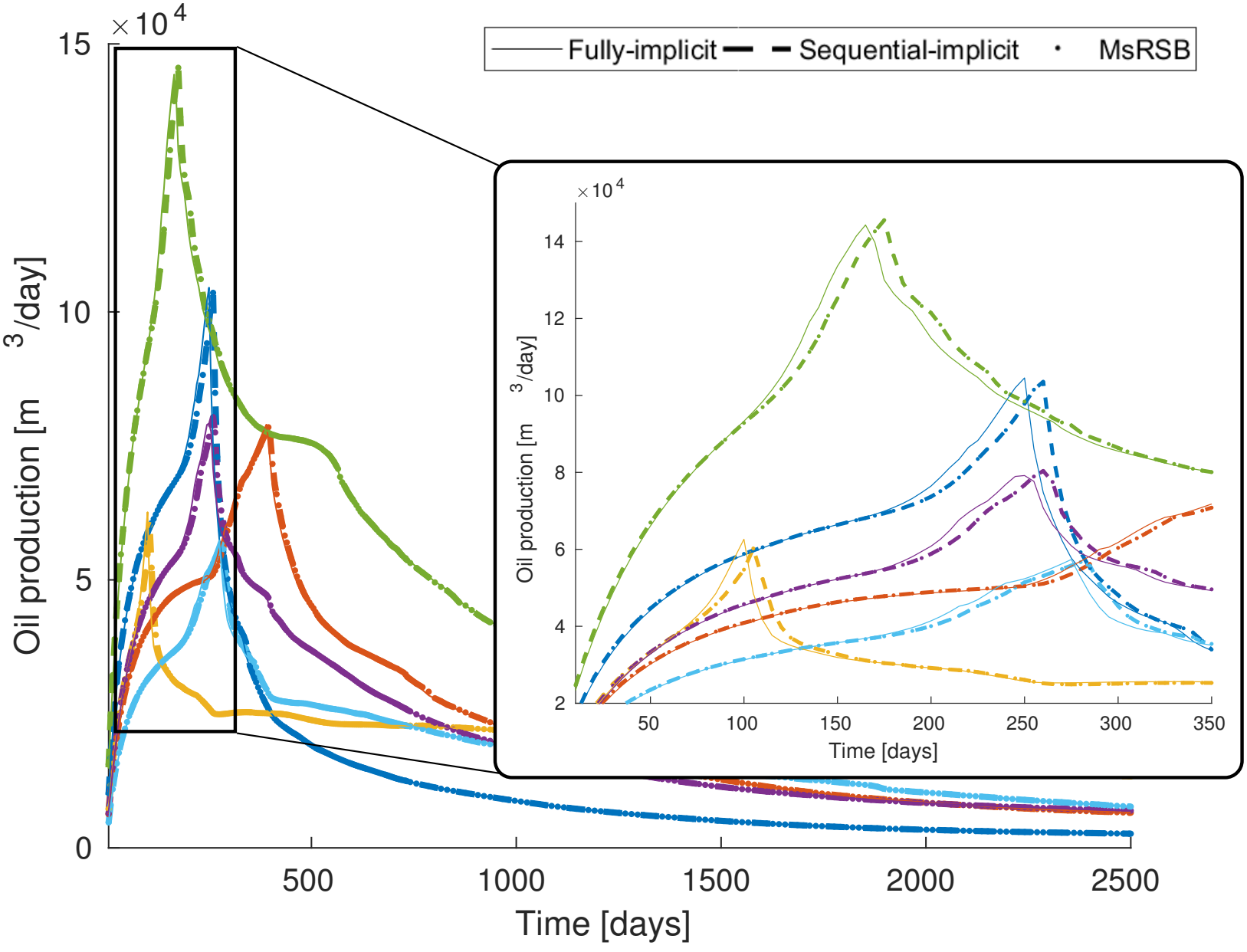
Example: Multiscale Norne



Example: Multiscale Norne



Example: Multiscale Norne

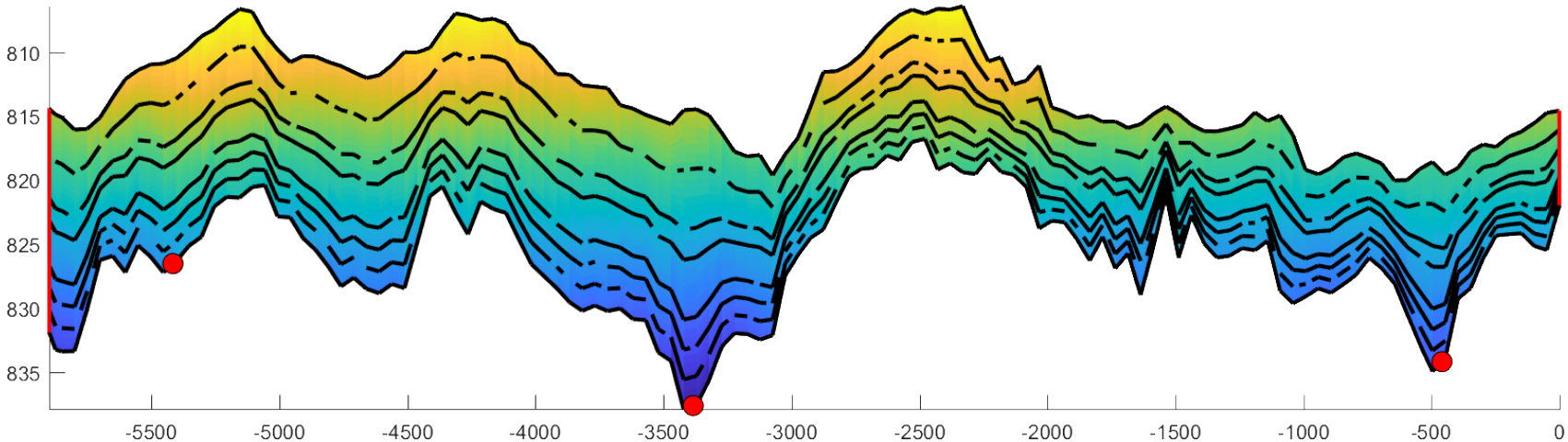


Outline

Introduction

Simulation of CO_2 storage requires vast temporal and spatial scales

- Vertical-equilibrium (VE) models: Accurate and efficient in this regime
- Assumption of vertical segregation is not always valid
- Near-well regions, layered flow compartments, coupling to facilities, ...

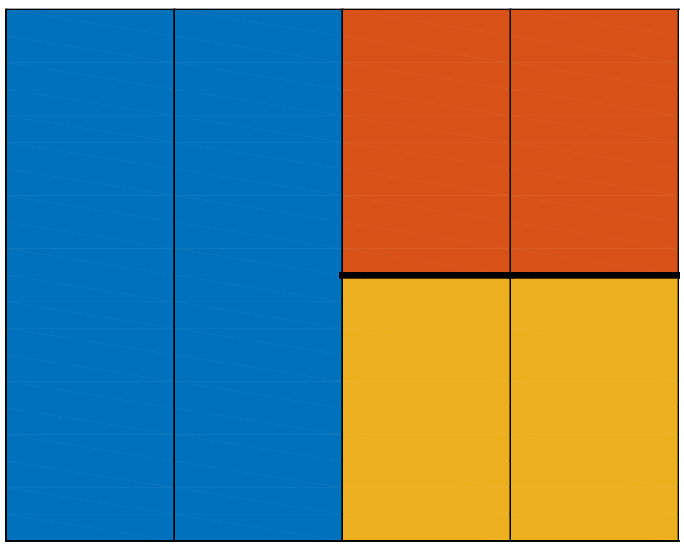


Idea: *Combine multiple discretization regions in a robust, unified model*
Joint work with Halvor Møll Nilsen

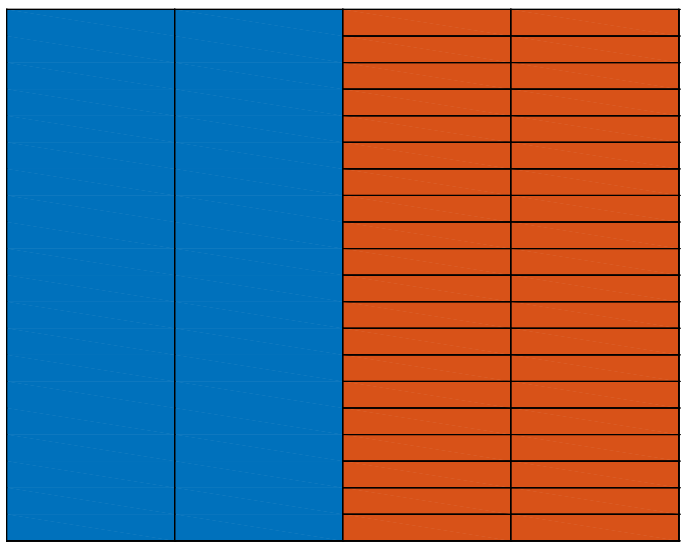
Introduction

Fully-implicit, finite-volume discretization

- Fully automated coarsening for e.g. corner-point grids
- Regions are automatically detected and discretized
- Challenge: Consistent coupling between different regions
- Transition between VE-zones, diffuse leakage, fine-scale are all included

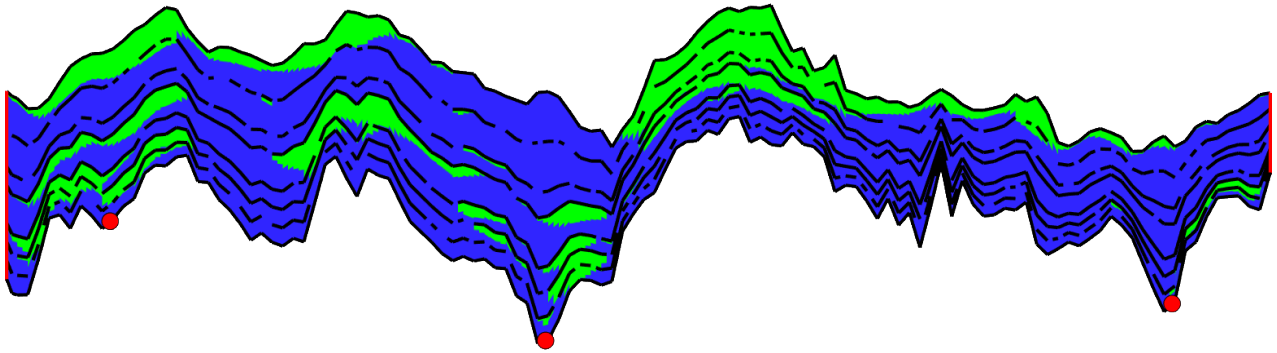


VE to VE

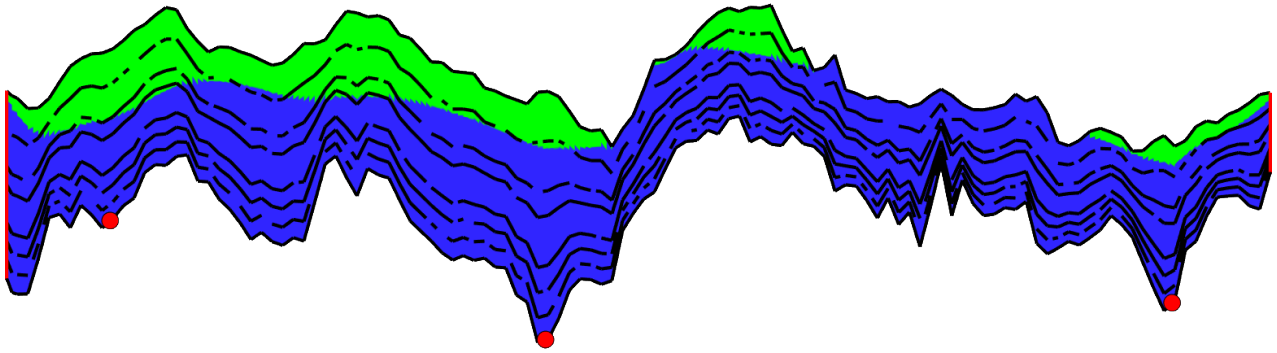


Fine-scale to VE

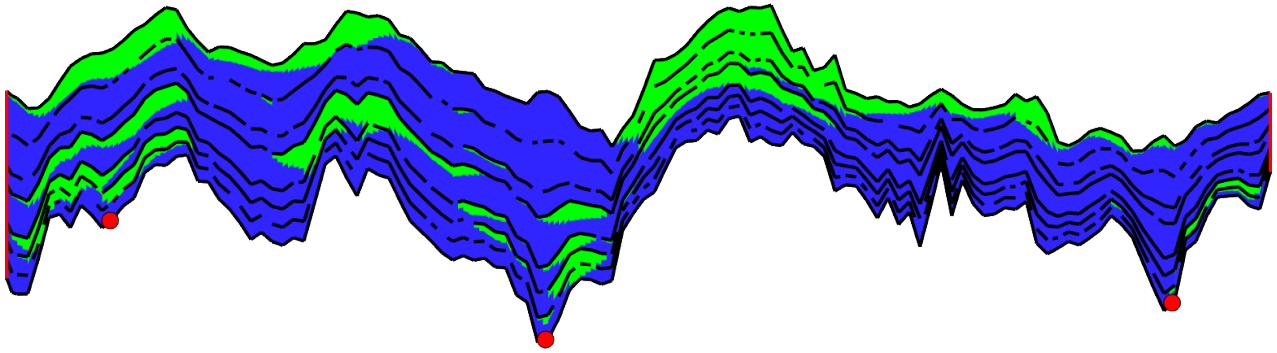
Example: Sleipner (injection)



Fine-scale (40,460 cells)

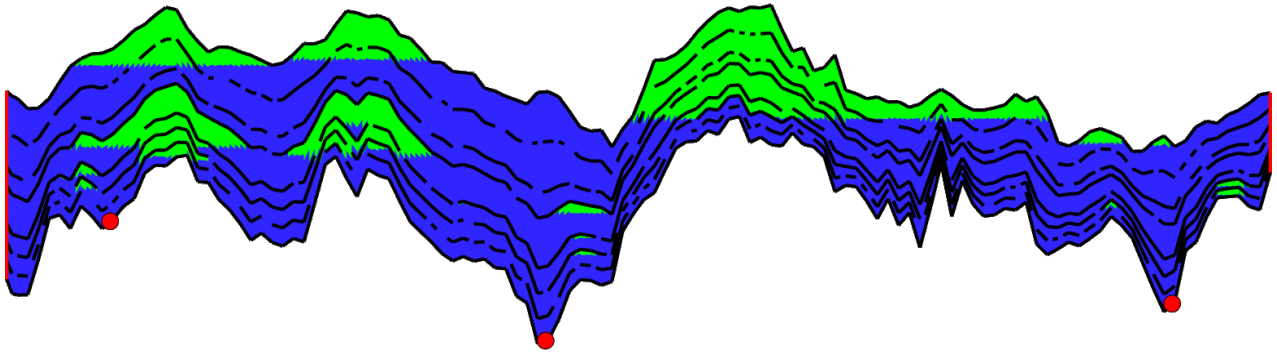


VE (700 cells)

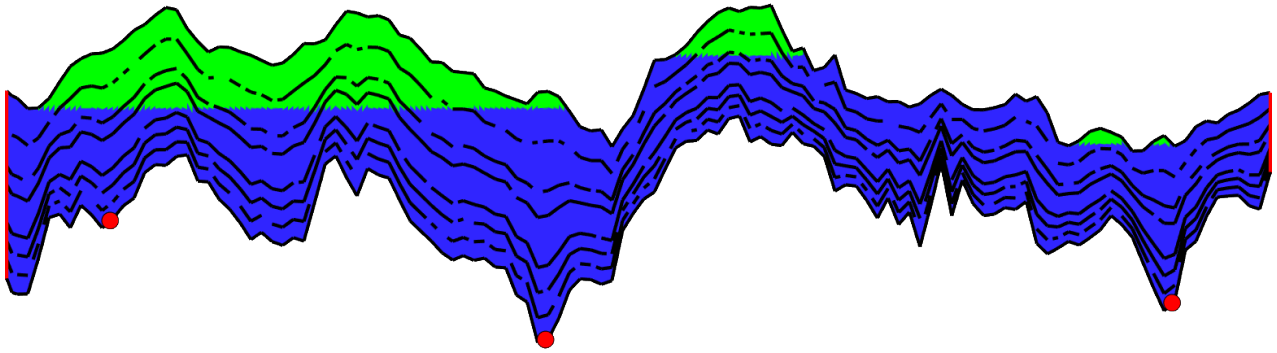


Hybrid (2700 cells)

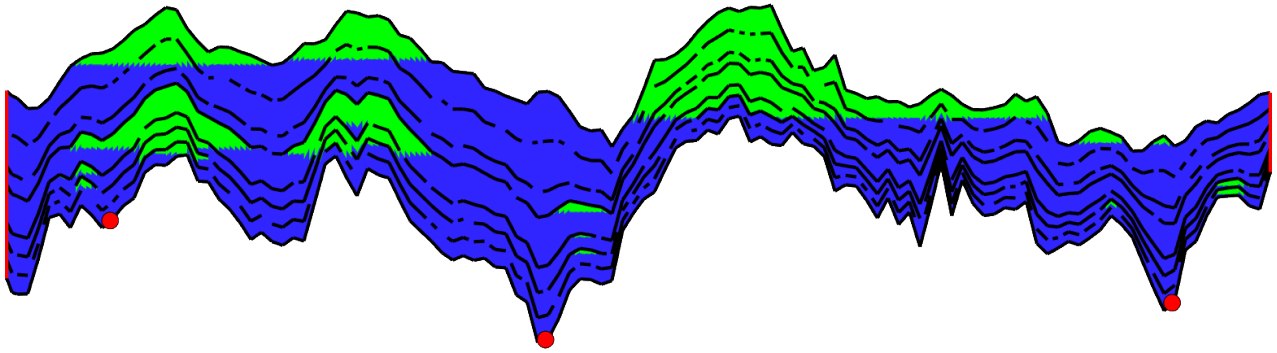
Example: Sleipner (migration)



Fine-scale (40,460 cells)

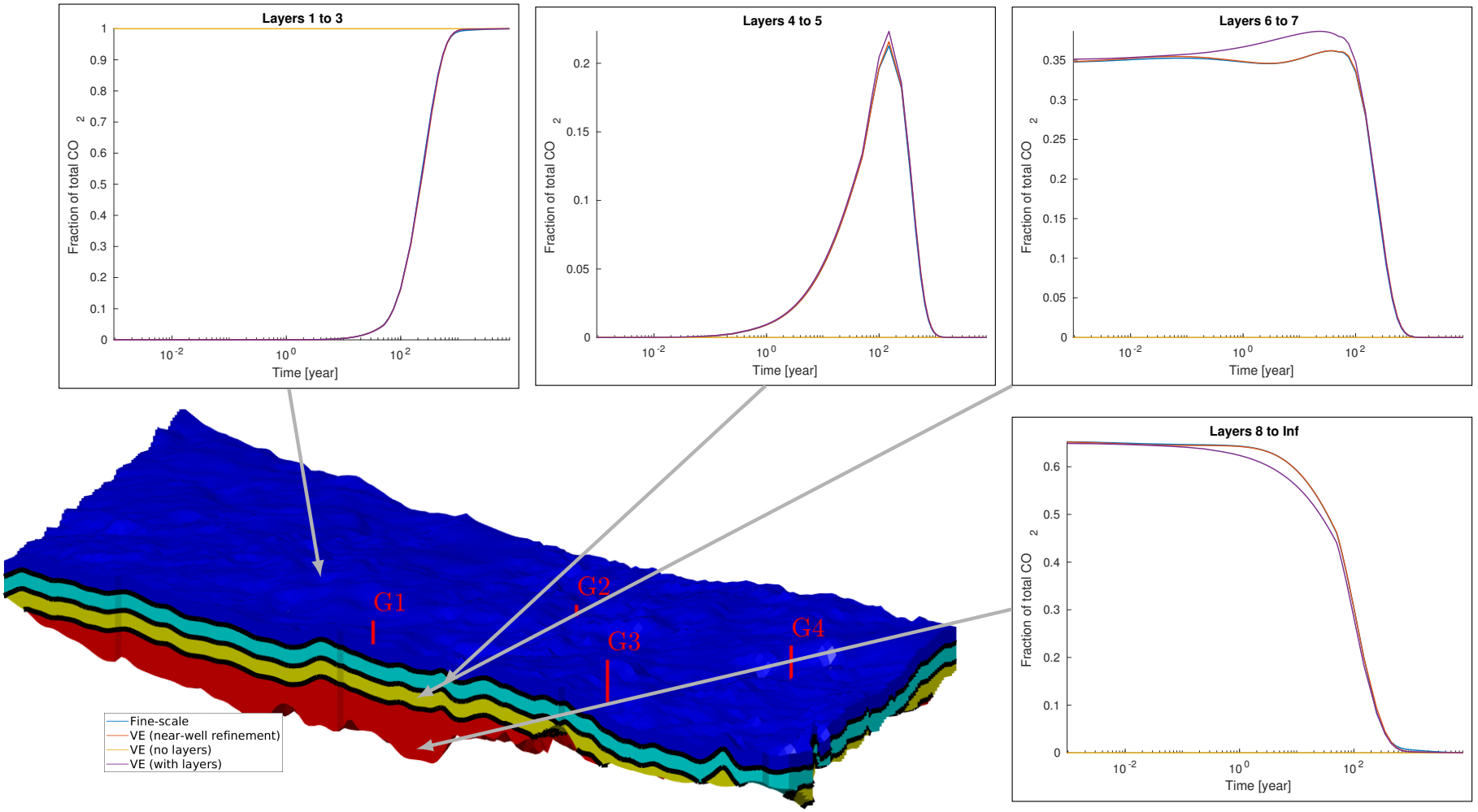


VE (700 cells)



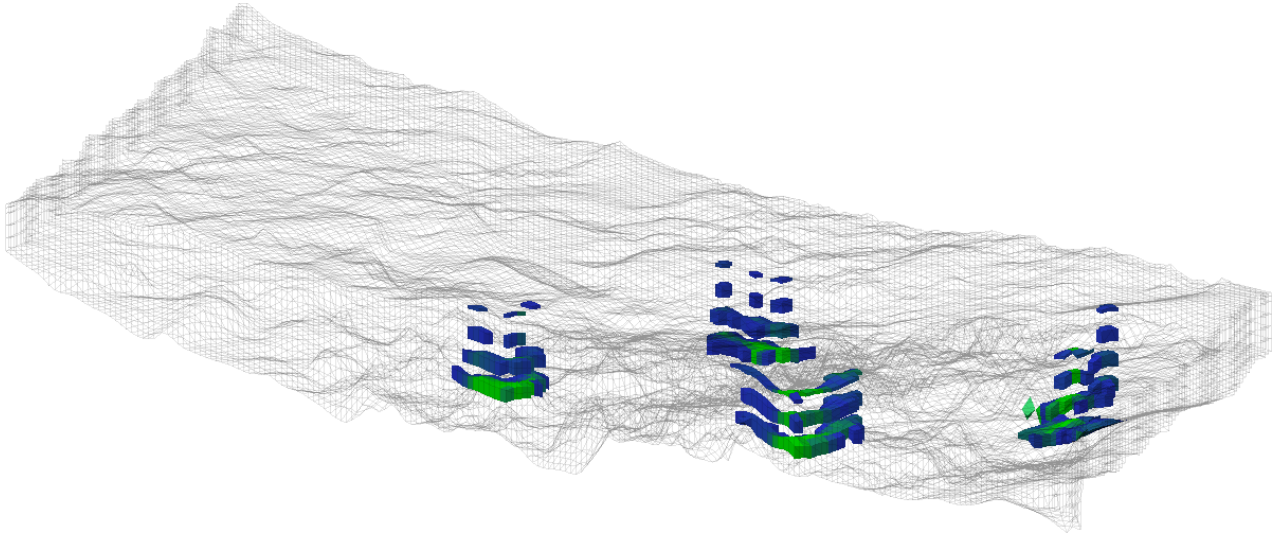
Hybrid (2700 cells)

Example: Full Utsira model

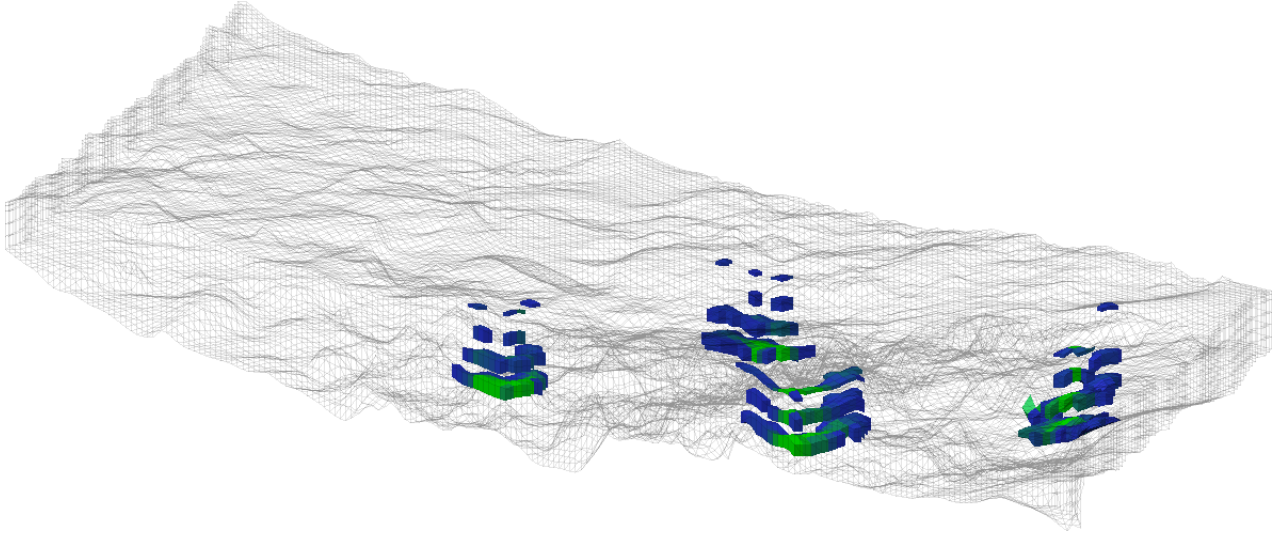


Utsira model provided by the Norwegian Petroleum Directorate

Example: Utsira, (migration)

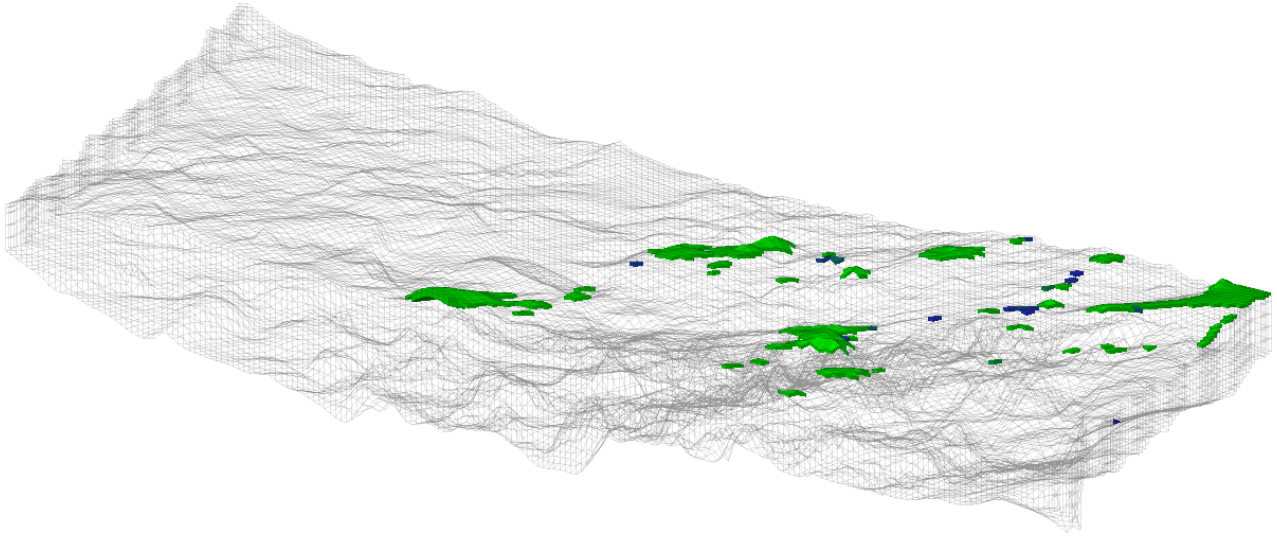


Fine-scale (111,162 cells)

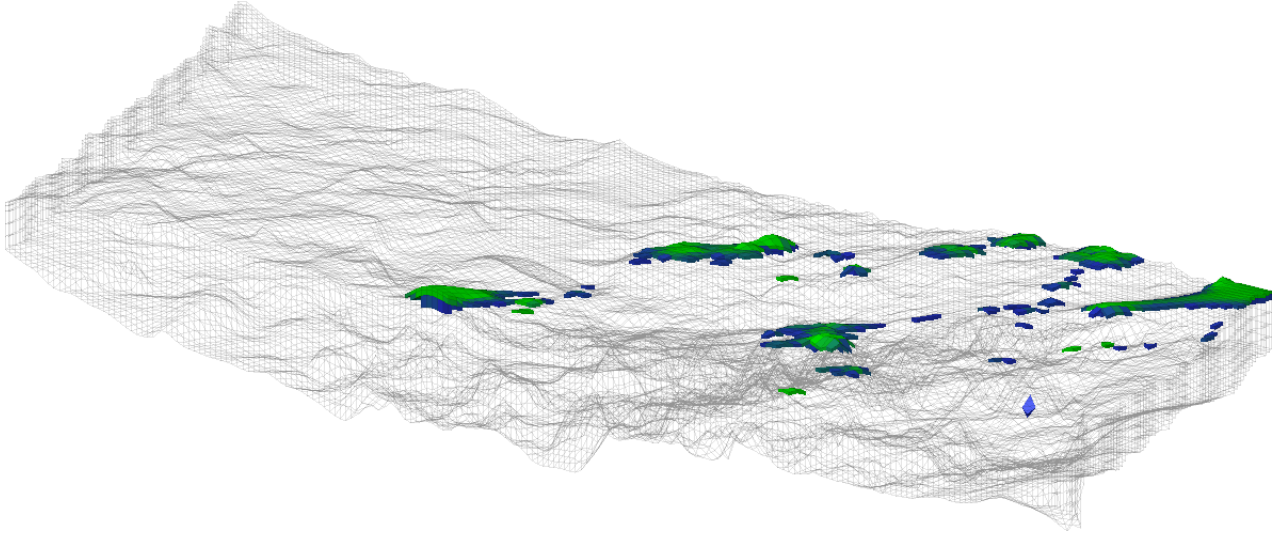


Hybrid (44,215 cells)

Example: Utsira (migration)

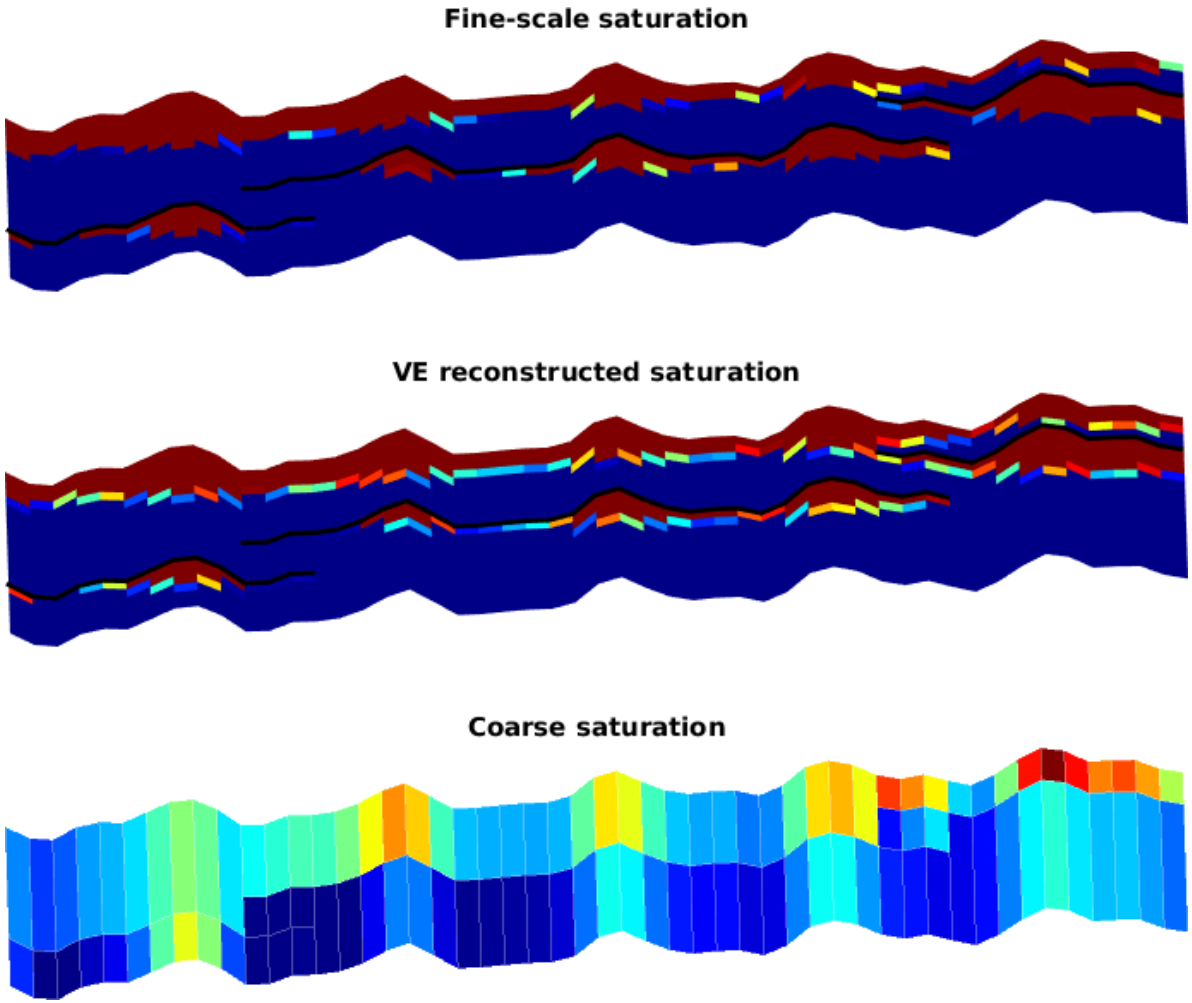


Fine-scale (111,162 cells)



Hybrid (44,215 cells)

Combining models



Compositional VE-hybrid model in five lines of code!

Overview: core and add-on modules