JutulDarcy.jl

A fully differentiable high-performance reservoir simulator based on automatic differentiation

Olav Møyner, SINTEF Digital, Applied computational sciences

ECMOR 2024, September 3rd, 2024

Motivation: Why a new code?

SINTEF is a contract research institute with many simultaneous projects, software is used in many projects with different needs and requires rapid turnaround

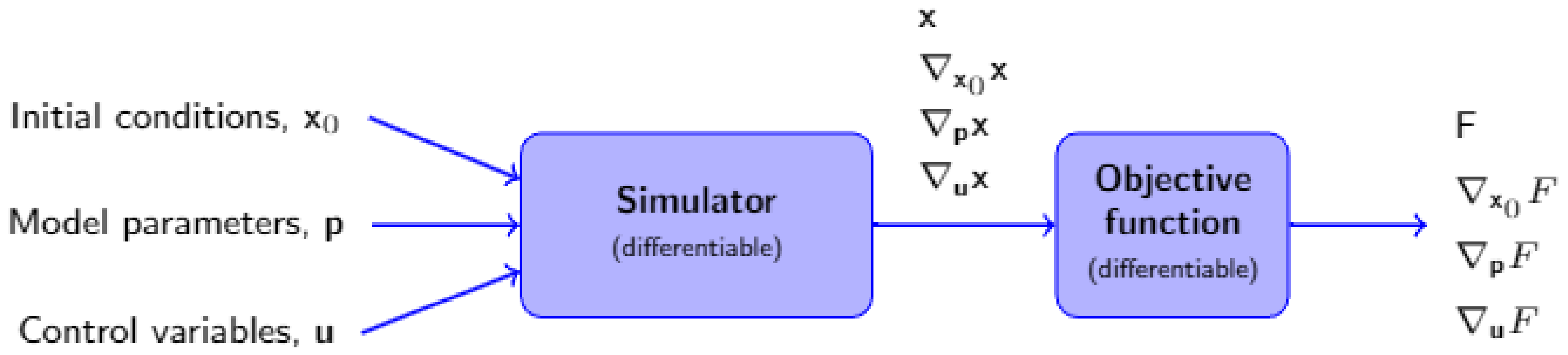
MRST is written in MATLAB/Octave and uses automatic differentiation (AD)

- Code was primarily developed for reservoir simulation, but is used for many projects
- Examples: Batteries, fuel cells, electrolysis, CO2 capture processes, …
- Issues with MATLAB for new applications:
  - Poor performance on hard-to-vectorize and small models
  - Commercial license required
  - Difficult to deploy for clients
  - Most universities have transitioned to Python (with some Julia)

  Most new models are heterogenous "multi-physics" models you do not have a single large component that can be optimized for performance (no big "reservoir" that is biggest assembly cost)

Motivation: Reservoir simulation

- Reservoir simulators are typically used with standard input files and standard output
- For some workflows this is a bad fit – want to "script" the simulator
- A fully differentiable simulator is not useful if it is constrained to a hard-to-build C++ code or derivatives are only accessible via IO


Initial conditions, $\mathbf{x}_0$
Model parameters, $\mathbf{p}$
Control variables, $\mathbf{u}$
Simulator
(differentiable)
$\mathbf{x}$
$\nabla_{\mathbf{x}_0}\mathbf{x}$
$\nabla_{\mathbf{p}}\mathbf{x}$
$\nabla_{\mathbf{u}}\mathbf{x}$
Objective function
(differentiable)
F
$\nabla_{\mathbf{x}_0}F$
$\nabla_{\mathbf{p}}F$
$\nabla_{\mathbf{u}}F$

Taken from Odd Andersen's presentation yesterday

Motivation: Types of codes

Research efficiency is the time taken to arrive at a conclusion

= modelling time + simulation time + paper/report writing time

Type of code	Typical language	Simulation performance	Ease of setup	Ease of modification
HPC code	Fortran, C++	Really fast – and scales! 🙂	A few days fighting CMake 😐	~O(PhD duration) 🙁
Commercial code	You get a binary...	Not great, not terrible 😐	Quick and costly 😐	Impossible 💀
Flexible code	Python, MATLAB	Slow – hard to scale 🙁	Quick and free 🙂	Easy 🙂

Motivation: Software engineering

- High level, flexible codes eventually hit a performance wall
  - You start writing extensions in C, C++ or Fortran
  - Your nice self-contained code bundle now requires a build system and careful memory management
- Low-level, high-performance codes eventually hit a flexibility wall
  - You start adding a high-level Python or MATLAB API to control your simulator
  - Difficult to translate a HPC simulator memory and execution model to a high-level API that is useful

High risk of getting the worst of both worlds!

Machine learning: Libraries have a high-level Python layer and a high-performance layer, with a **substantial engineering effort** to make deployment and install easy

Something to ponder: Would there be less papers presented on machine learning if reservoir simulators were as easy to use and as accessible as for example PyTorch?

Motivation: Extending models with new effects

Two physical models

Model A with linearization

$$\mathbf{R}_a(\mathbf{x}_a) = \mathbf{0}, \quad -\mathbf{J}_{aa}\Delta\mathbf{x}_a = -\mathbf{R}_a$$

Model B with linearization

$$\mathbf{R}_b(\mathbf{x}_b) = \mathbf{0}, \quad -\mathbf{J}_{bb}\Delta\mathbf{x}_b = -\mathbf{R}_b$$

Combined model

$$R(\mathbf{x}_a, \mathbf{x}_b) = \begin{bmatrix} R_a(\mathbf{x}_a, \mathbf{x}_b) \\ R_b(\mathbf{x}_b, \mathbf{x}_a) \end{bmatrix} = \mathbf{0}$$

$$\begin{bmatrix} \mathbf{J}_{aa} & \mathbf{J}_{ab} \\ \mathbf{J}_{ba} & \mathbf{J}_{bb} \end{bmatrix} \begin{bmatrix} \mathbf{x}_a \\ \mathbf{x}_b \end{bmatrix} = \begin{bmatrix} \mathbf{R}_a \\ \mathbf{R}_b \end{bmatrix}$$

Models of interest consist of *many* sub-models. Need *fast* automatic differentiation!

Motivation: Building blocks of the adjoint method

$$\mathbf{R}_i\left(\mathbf{x}_i(\mathbf{p}), \mathbf{x}_{i-1}(\mathbf{p}), \mathbf{p}, \mathbf{f}_i\right) = 0, \quad \forall i \in \{1, \dots, n\}$$

$$\lambda_i = -\left(\frac{\partial \mathbf{R}_i}{\partial \mathbf{x}_i}^T\right)^{-1}\left(\frac{\partial O_i}{\partial \mathbf{x}_i}^T + \frac{\partial \mathbf{R}_{i+1}}{\partial \mathbf{x}_i}^T \lambda_{i+1}\right) \qquad \frac{dJ_\lambda}{d\mathbf{p}} = \sum_{i=1}^{n}\left[\frac{\partial O_i}{\partial \mathbf{p}}^T + \frac{\partial \mathbf{R}_i}{\partial \mathbf{p}}^T \lambda_i\right]^T = \frac{dO}{d\mathbf{p}}$$

From forward simulation

$\left(\frac{\partial \mathbf{R}_i}{\partial \mathbf{x}_i}\right)^T$ Linear solve

$\left(\frac{\partial \mathbf{R}_{i+1}}{\partial \mathbf{x}_i}\right)$ Matrix-vector product

New shape, but uses internals only

$\left(\frac{\partial \mathbf{R}_i}{\partial \mathbf{p}}\right)^T$ Matrix-vector product

User provided code!

$\left(\frac{\partial O_i}{\partial \mathbf{x}}\right)^T$ Element-wise addition

$\left(\frac{\partial O_i}{\partial \mathbf{p}}\right)^T$ Element-wise addition

See: *Adjoint-based optimization of multi-phase flow through porous media - a review* by J. D. Jansen, 2011

SINTEF

Fully differentiable simulators

Goal: A fully differentiable, scriptable, high-performance porous media simulator

1. Does not need to be *the* fastest, but must be *competitive*
2. Does not need to run *all* models, but must *run relevant models*
3. Try to build on experiences from AD in AD-GPRS, MRST and OPM Flow

Many alternatives:

- Python with accelerators: Numba/JAX and in the future Mojo?
- Bolt something onto machine learning libraries (PyTorch/TensorFlow)?
- Modern C++/Fortran?
- Adapt existing codes?
- New languages: Chapel, Rust, Carbon, Julia?

Main considerations:

- Native support for automatic differentiation for coupled models
- Interactive development without big compilation toolchain
- Maturity (per 2020) and available building blocks:

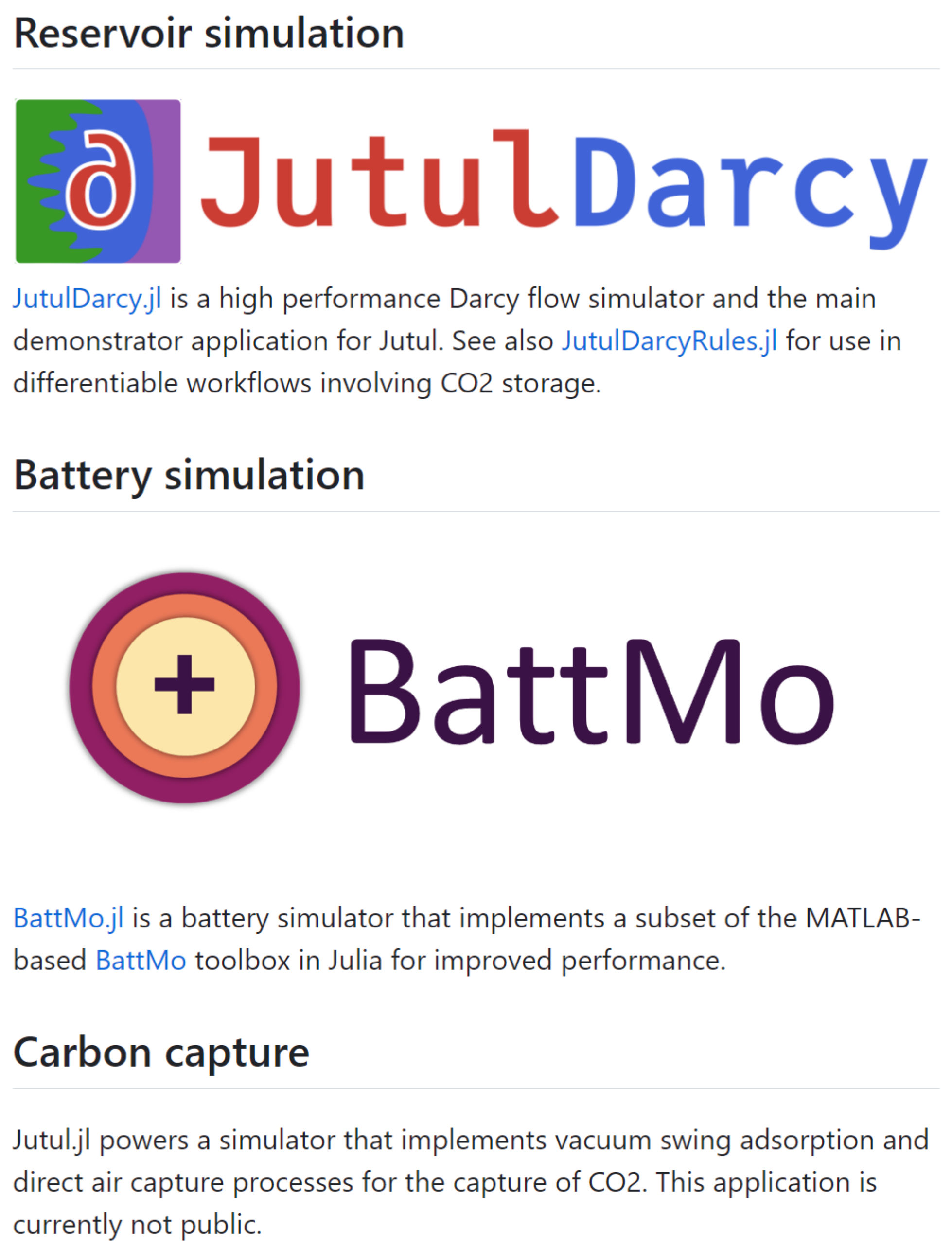
→ julia

SINTEF

Jutul

- Julia package for implicit solves of coupled models
- Automatic differentiation of discrete equations
- Design inspired by use of AD in MRST and OPM Flow
- Robust Newton solvers (chopping, relaxation, variable scaling, absolute and relative change limits)
- Interactive visualization

www.github.com/sintefmath/Jutul.jl



Reservoir simulation

JutulDarcy

JutulDarcy.jl is a high performance Darcy flow simulator and the main demonstrator application for Jutul. See also JutulDarcyRules.jl for use in differentiable workflows involving CO2 storage.

Battery simulation

BattMo

BattMo.jl is a battery simulator that implements a subset of the MATLAB-based BattMo toolbox in Julia for improved performance.

Carbon capture

Jutul.jl powers a simulator that implements vacuum swing adsorption and direct air capture processes for the capture of CO2. This application is currently not public.

SINTEF

JutulDarcy demonstrator

Main demonstrator is a porous media simulator:

- Immiscible, thermal, black-oil and equation-of-state or K-value compositional flow
- Industry standard input – or write scripts where you define grid, properties and schedule yourself
- *Fully differentiable*: Gradients with respect to any declared parameter using adjoint method
- MPI parallel with BoomerAMG and threads
- MIT licensed, open source at www.github.com/sintefmath/JutulDarcy.jl
- Installation on any OS (Julia 1.8+):

```
using Pkg; Pkg.add("JutulDarcy")
```

JutulDarcy

Compact implementation
JutulDarcy: 9000 lines of code (loc)

- **Immiscible flow base 1000 loc**
- **Black-oil specialization 1100 loc**
- **Multisegment wells with advanced control logic 1430 loc**
- **Compositional 705 loc**

Correspondence between equations and code

- All equation terms are written as functions without type annotations or templates
- Dispatch is used to distinguish different types of physics and discretizations
- Equations are easily recognizable in code

$$\boxed{\begin{array}{l} \frac{\partial}{\partial t}[\phi \rho_o^s (b_o S_o + R_s b_g S_g)] \\ \frac{\partial}{\partial t}[\phi \rho_g^s (b_g S_g + R_v b_o S_o)] \end{array}} \begin{array}{l} + \rho_o^s \nabla \cdot (b_o \vec{v}_o + R_s b_o \vec{v}_o) - q_o = 0, \\ + \rho_g^s \nabla \cdot (b_g \vec{v}_g + R_v b_g \vec{v}_o) - q_g = 0 \end{array}$$

```julia
function update_blackoil_mass!(M, pv, b, S, Rs, Rv, rhoS, sys)
    Φ = pv[cell]                              # pore-volume
    l, v = phase_indices(sys)                 # 1, 2 or 2, 1
    bO, bG = b[l, cell], b[v, cell]           # get b-factors
    sO, sG = S[l, cell], S[v, cell]           # get saturations
    M[l] = Φ*rhoS[l]*(bO*sO + bG*sG*Rv[cell]) # oil component mass
    M[v] = Φ*rhoS[v]*(bG*sG + bO*sO*Rs[cell]) # gas component mass
    return M
end
```

Example: Black-oil flux implementation

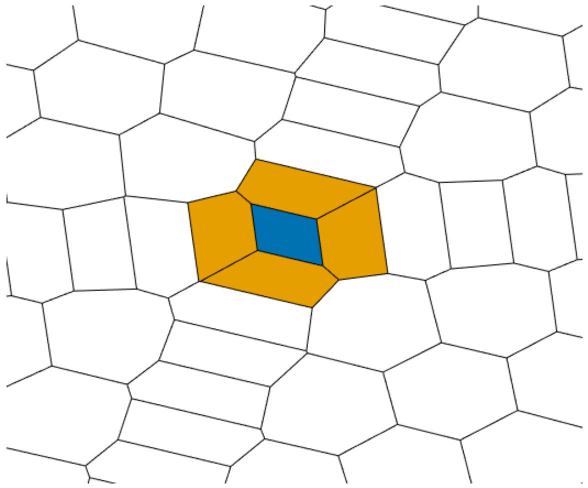
```
# Oil
f_bl = cell -> b_mob(b, kr, μ, l, cell)
λb_l = upwind(upw, f_bl, ψ_l)
q_l = rhoS[l]*λb_l*ψ_l
# Gas mobility
f_bv = cell -> b_mob(b, kr, μ, v, cell)
λb_v = upwind(upw, f_bv, ψ_v)
# Rs (solute gas) upwinded by liquid potential
f_rs = cell -> @inbounds Rs[cell]
rs = upwind(upw, f_rs, ψ_l)
# Final flux = gas phase flux + gas-in-oil flux
q_v = (λb_v*ψ_v + rs*λb_l*ψ_l)*rhoS[v]
```

```
function b_mob(b, kr, μ, ph, c)
    λ = kr[ph, c]/μ[ph, c]
    b_f = b[ph, c]
    return λ*b_f
end
```

```
function upwind(upw::SPU, F, q)
    flag = q >= 0
    if flag
        up = upw.right
    else
        up = upw.left
    end
    return F(up)
end
```

SINTEF

Fast automatic differentiation



Jutul.jl uses a tailored approach automatic differentiation that:

- Supports complex sparsity patterns (unstructured grids and complex wells)
- Allows for equations and variables on different entities (cells, faces, wells, ...)
- Key ideas:
  - Detect sparsity pattern of equations at start of simulation
  - Allocate dense memory for storage of dual numbers, in the order of evaluation
  - Manage properties in a precomputed dependency graph
  - Avoid (significant) heap allocation during simulation and make GC happy

General AD for any discrete equation (Wells, facilities, coupling terms) → **General finite-volume scheme (AvgMPFA, NTPFA...)** → **Two-point flux scheme**

Specialization and performance →

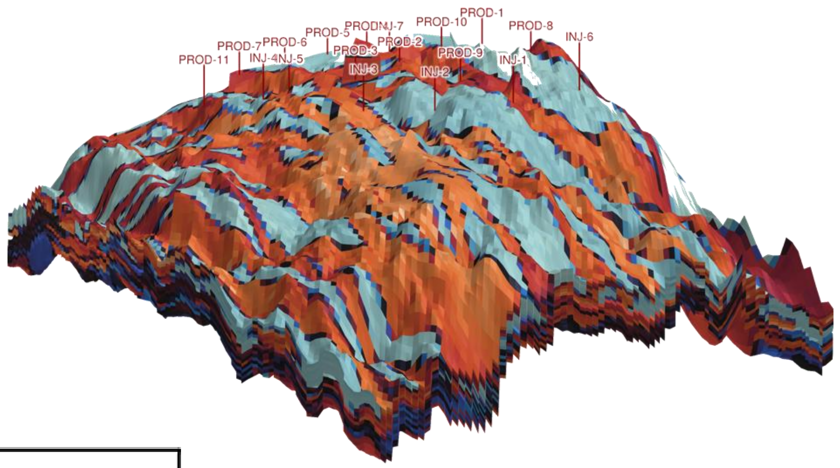
Case	Model	No. Cells	No. Wells	No. Assembly	Runtime	AD + props + assembly
SPE1	Black oil, 3-phase	300	2	521 (0.26 ms)	0.3 s	24 %
SPE9	Black-oil, 3-phase	9 000	26	403 (6.5 ms)	6.1 s	26 %
Egg	Water-oil, 2-phase	18 553	12	706 (4.9 ms)	15.8 s	11 %
Norne	Black-oil, 3-phase	44 431	36	2597 (41.5 ms)	260 s	18 %
Olympus	Water-oil, 2-phase	192 749	18	807 (62.3 ms)	162 s	18 %
Sleipner	Water-gas, 2-phase (CO2)	1 986 176	1	1965 (704 ms)	4 519 s	18 %
A	7 components, 3-phase	60 000	2	2097 (397 ms)	1071 s	43 %
B	Water-oil, 2-phase	200 000	100+	2299 (65 ms)	572 s	14 %
C	Black-oil, 3-phase	150 000	250+	18802 (132 ms)	10 205 s	16 %
D	Black-oil, 3-phase	1 250 000	100+	7259 (837 ms)	41 129 s	9.4 %

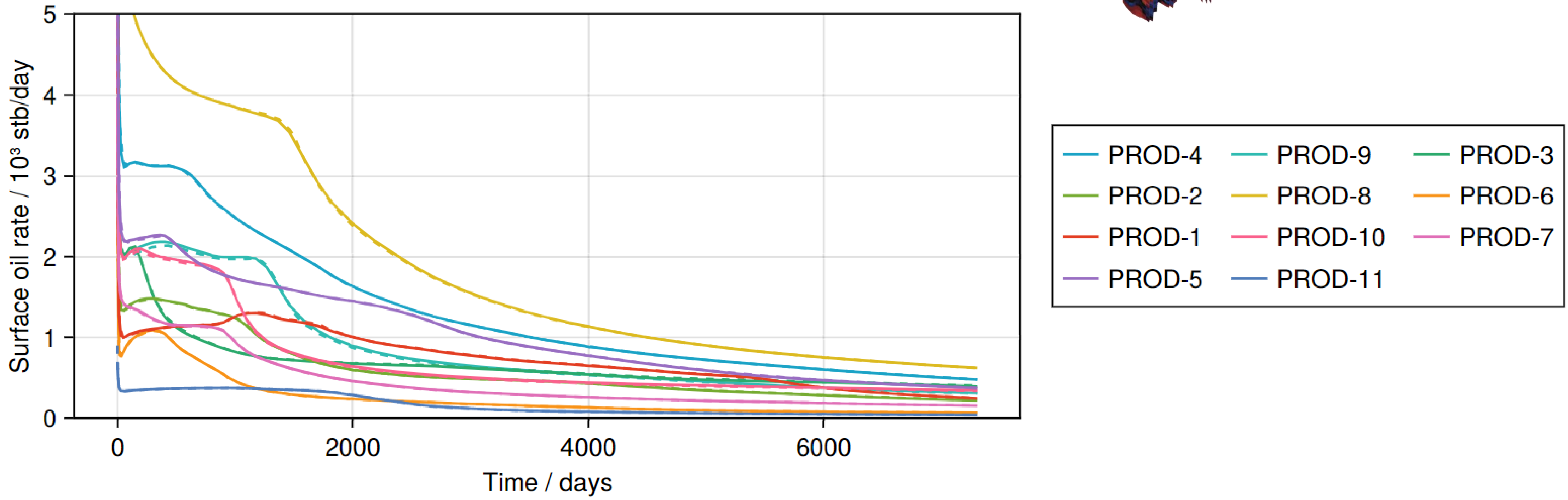
All cases run with –O3 single thread and default options on Ryzen 9 16 core CPU

Open models: Check against your favorite simulator!

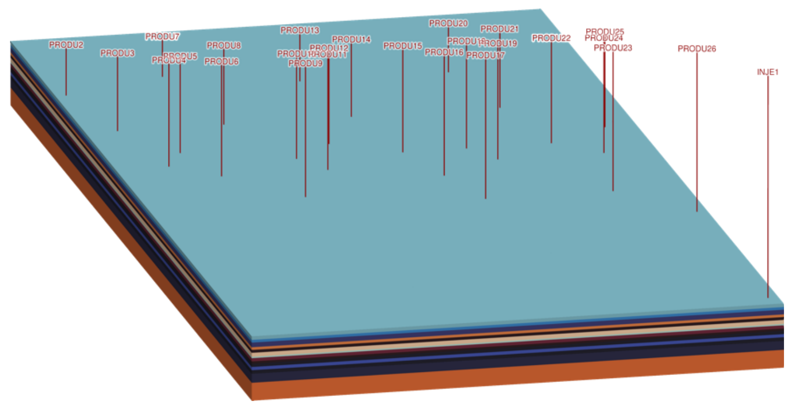
References: 10.2118/9723-PA, 10.3997/2214-4609.201802246, 10.2118/182679-PA 10.1002/gdj3.21, 10.2118/127538-MS, 10.2118/29110-MS, 10.2118/72469-PA

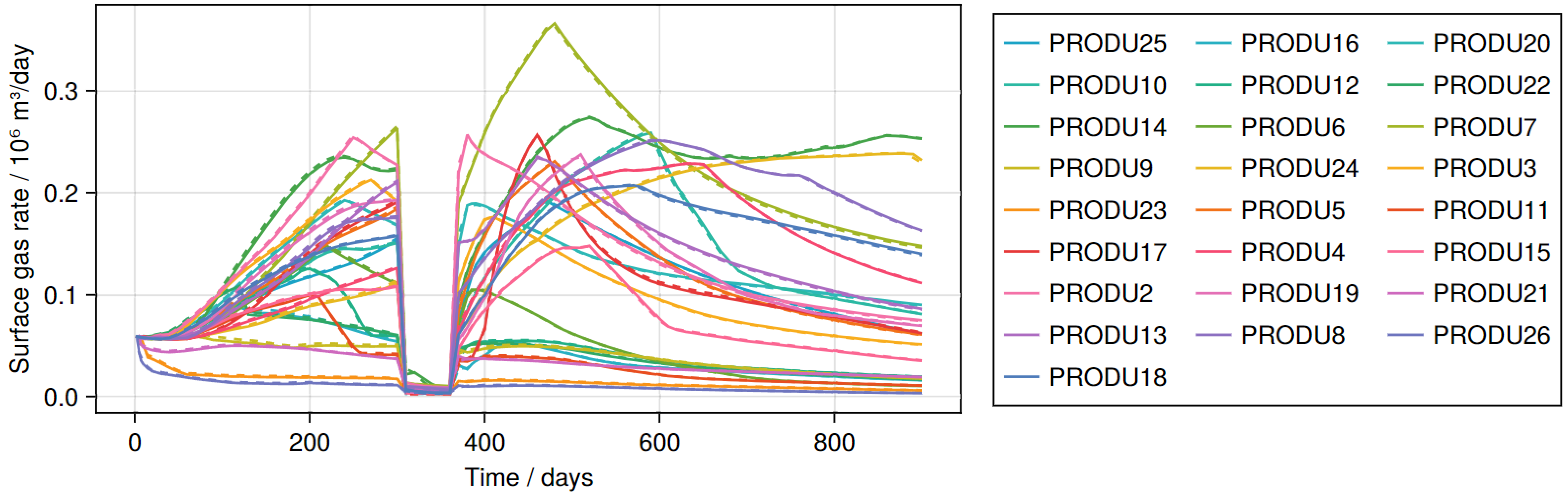
Validation: OLYMPUS 1



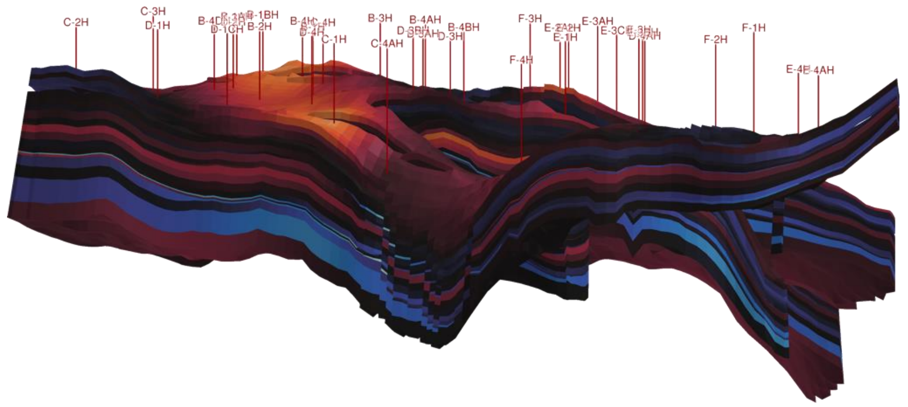


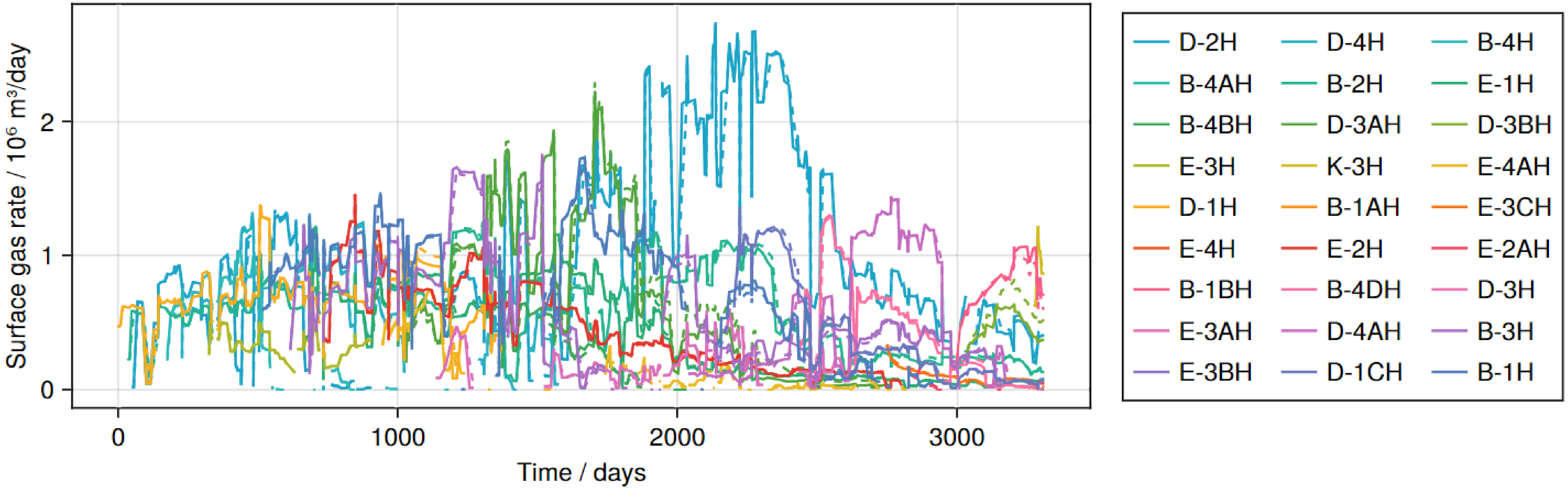
Verification: SPE9





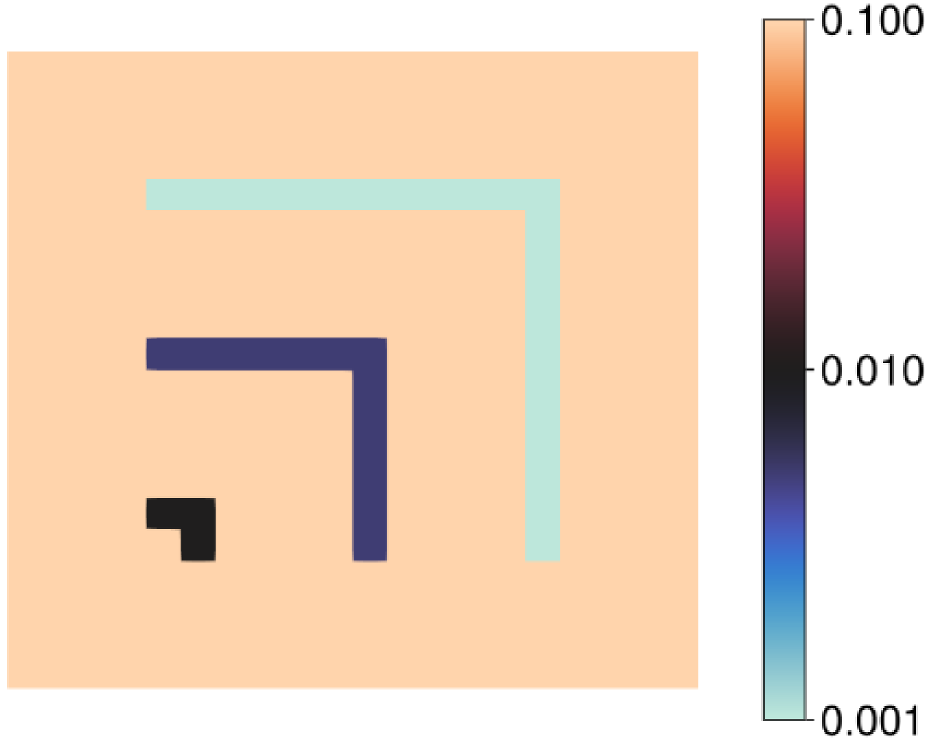
Validation: Norne





A toy example: Quarter-five with barriers

- Standard conceptual test case
- Gas is injected in bottom left and liquid is produced in upper right
- Layers with different rock types impede flow
- Viscous driven, no gravity
- How easy is it to modify a pre-existing model?



(a) Permeability (darcy)

Adding properties and parameters

Define new relative permeability defintion and evaluator function:

```julia
import JutulDarcy: AbstractRelativePermeabilities
struct MyKr <: AbstractRelativePermeabilities end
@jutul_secondary function update_my_kr!(vals, def::MyKr, model, Saturations,
    KrExponents, cells_to_update)
    for c in cells_to_update
        for ph in axes(vals, 1)
            S_α = max(Saturations[ph, c], 0.0)
            n_α = KrExponents[ph, c]
            vals[ph, c] = S_α^n_α
        end
    end
end
```

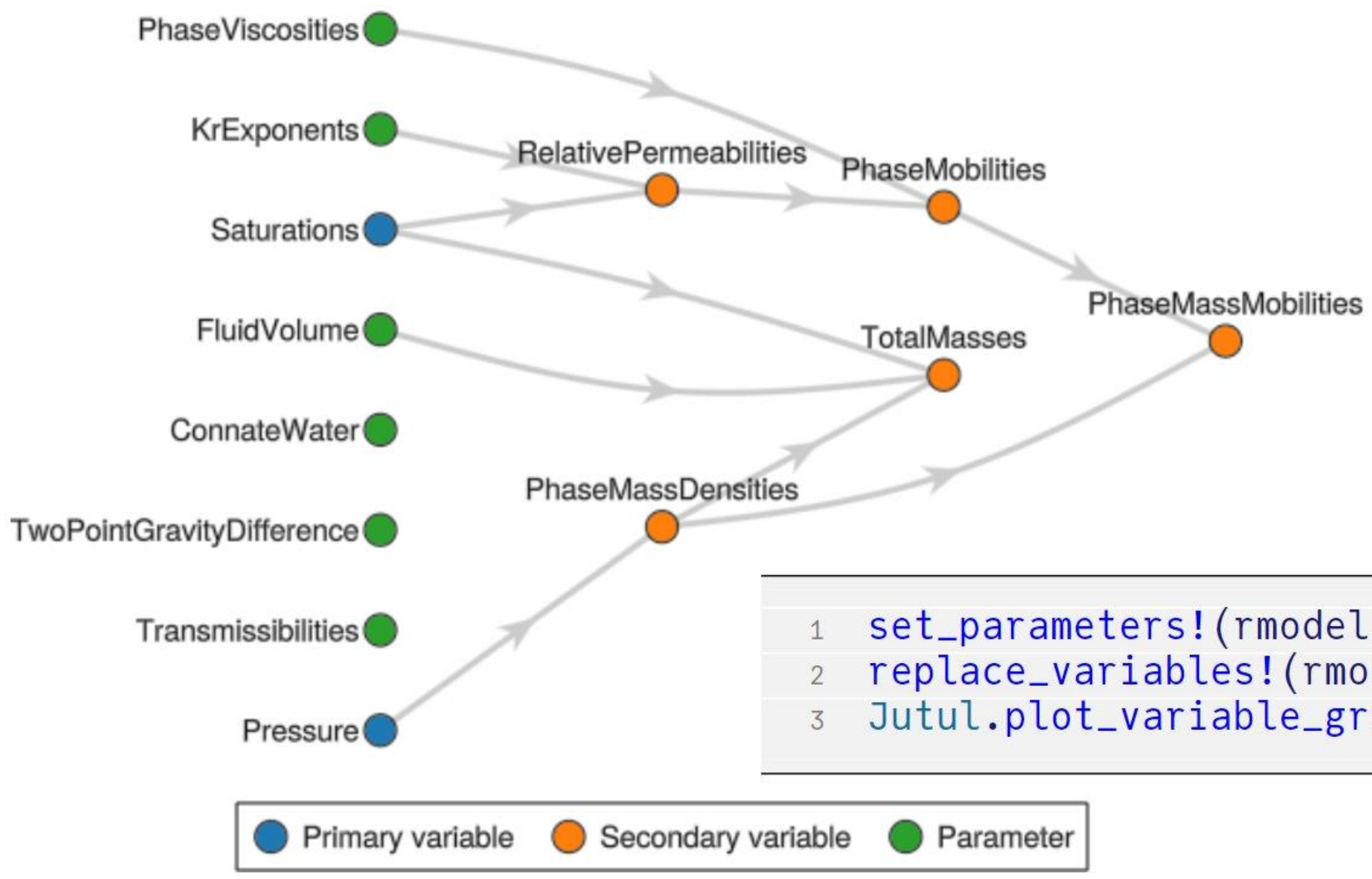
$$k_{rw} = S_\alpha^n, \; k_{rg} = S_\alpha^m$$

Exponents n, m vary spatially

Add a new parameter for Corey exponent:

```julia
import JutulDarcy: PhaseVariables
struct MyKrExp <: PhaseVariables end
Jutul.default_value(model, ::MyKrExp) = 2.0
```

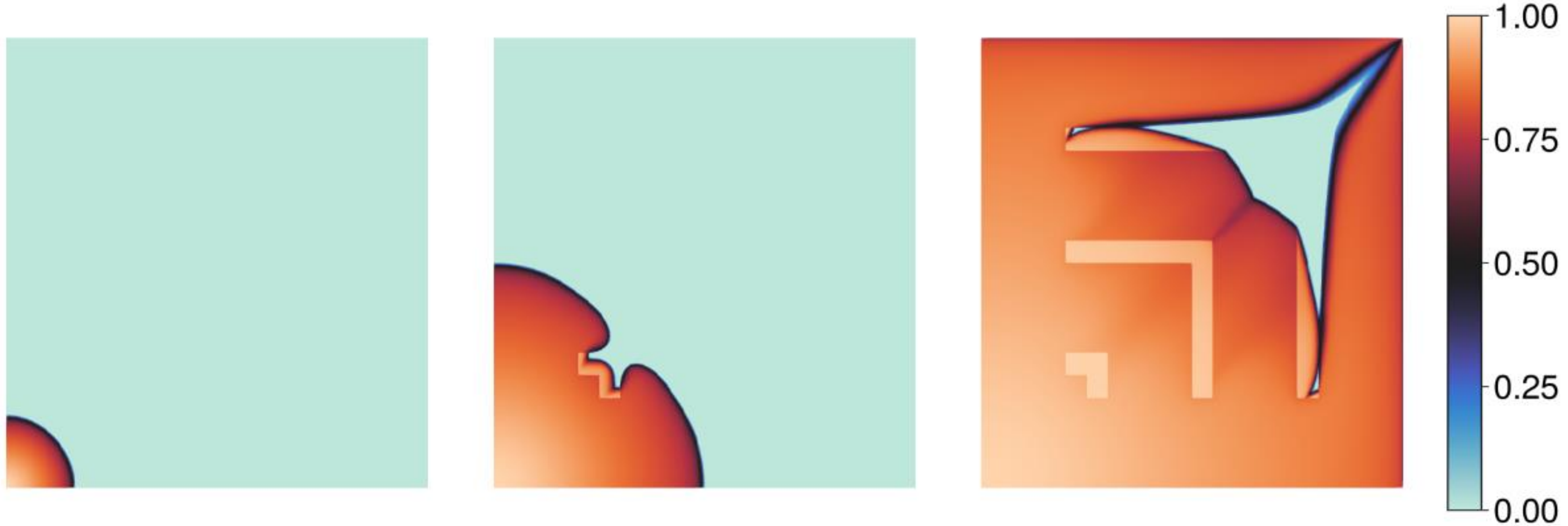
Dependency graph autogenerated!



```julia
set_parameters!(rmodel, KrExponents = MyKrExp())
replace_variables!(rmodel, RelativePermeabilities = MyKr())
Jutul.plot_variable_graph(rmodel)
```

Running the "new" simulator

- We set the values of the new parameter to vary with rock types
- Running a simulation immediately recompiles only small parts of the code
- Performance of new function is excellent – no difference in runtime after adding
- The inserted function could also be a neural network or a call to an external package



(b) Progression of gas saturation

SINTEF

Let us calculate some gradients!

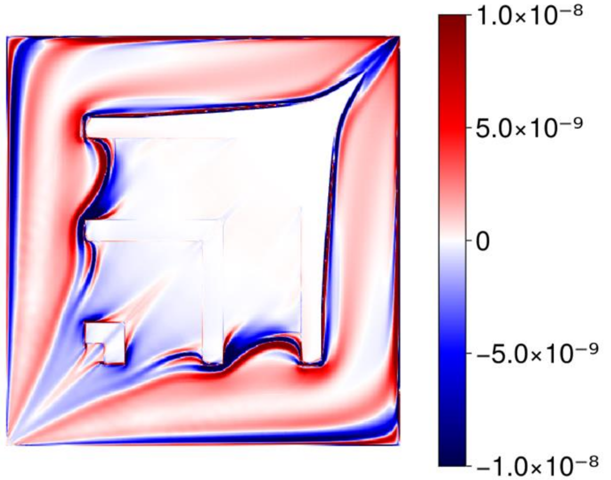
- Gradient calculations: Let us define the producer gas rate as objective

```julia
pv = pore_volume(model, prm)
total_time = sum(dt)
inj_rate = sum(pv)/total_time
import JutulDarcy: compute_well_qoi
function objective_function(model, state, Δt, step_i, forces)
    T = SurfaceGasRateTarget
    grat = compute_well_qoi(model, state, forces, :Producer, T)
    return Δt*grat/(inj_rate*total_time)
end
```

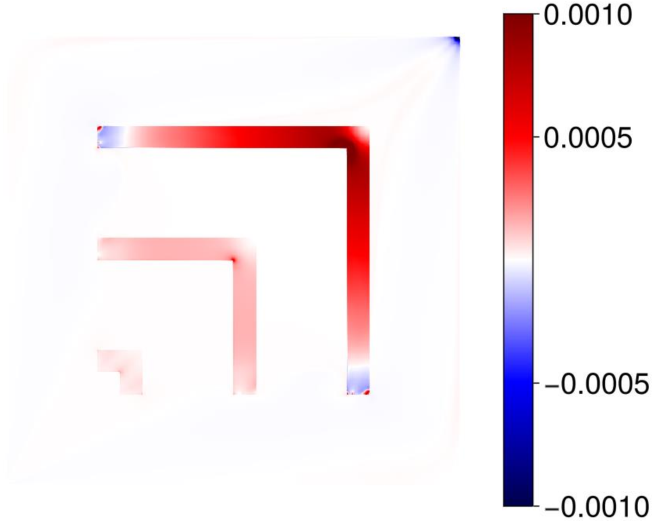
- Call high level interface to get gradients with respect to *input* parameters (perm, poro, geometry) rather than *numerical* parameters (transmissibilites and pore-volumes)

```julia
import JutulDarcy: reservoir_sensitivities
grad = reservoir_sensitivities(case, result, objective_function)
```

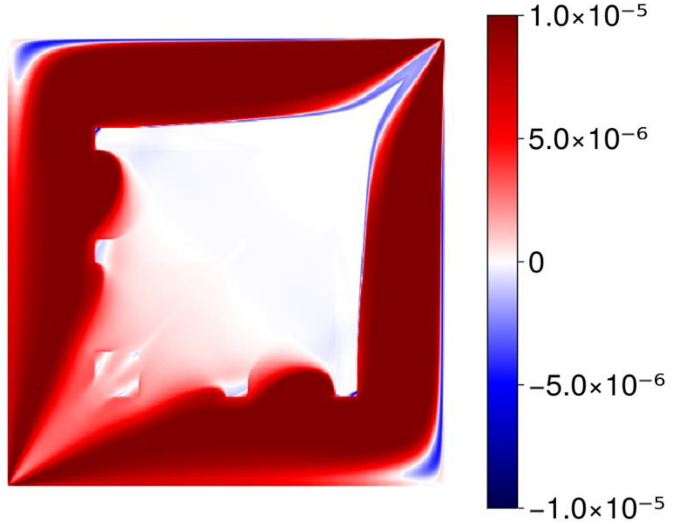
Examples of sensitivities from one adjoint solve



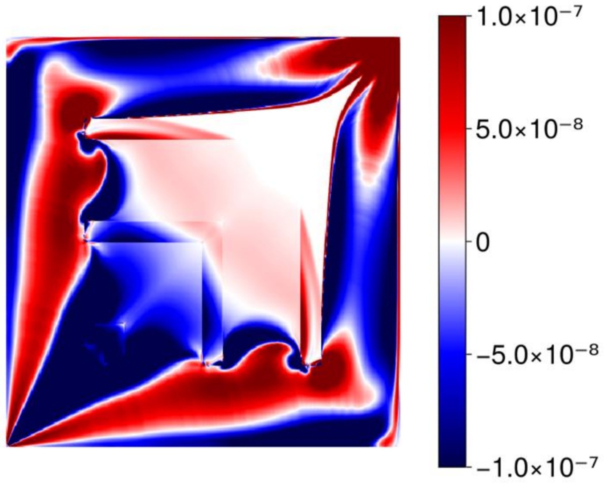
Cell depth



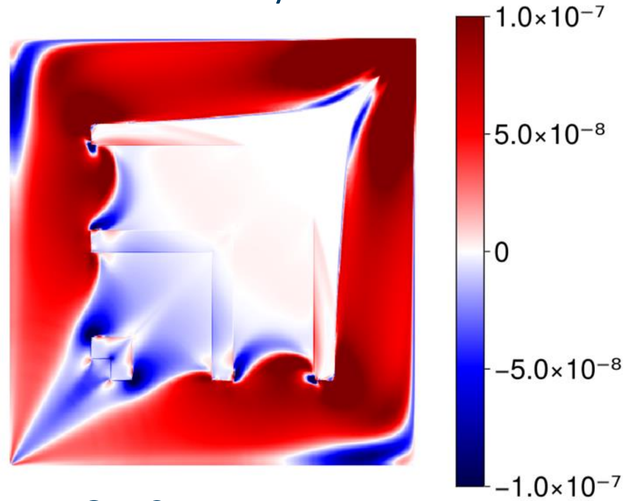
Permeability



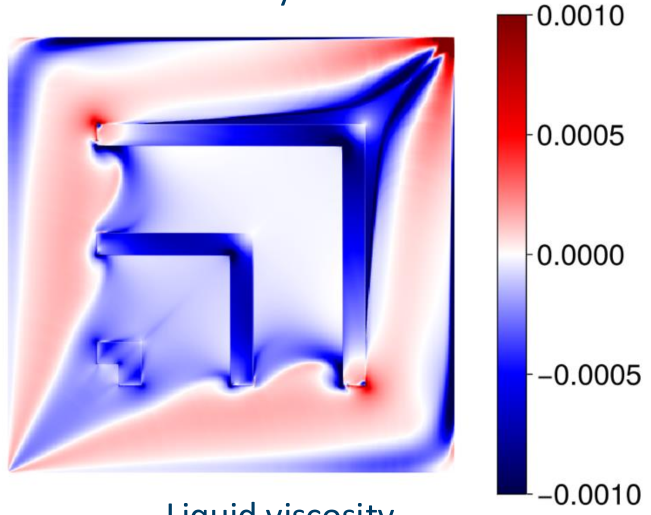
Porosity



Liquid Corey exponent



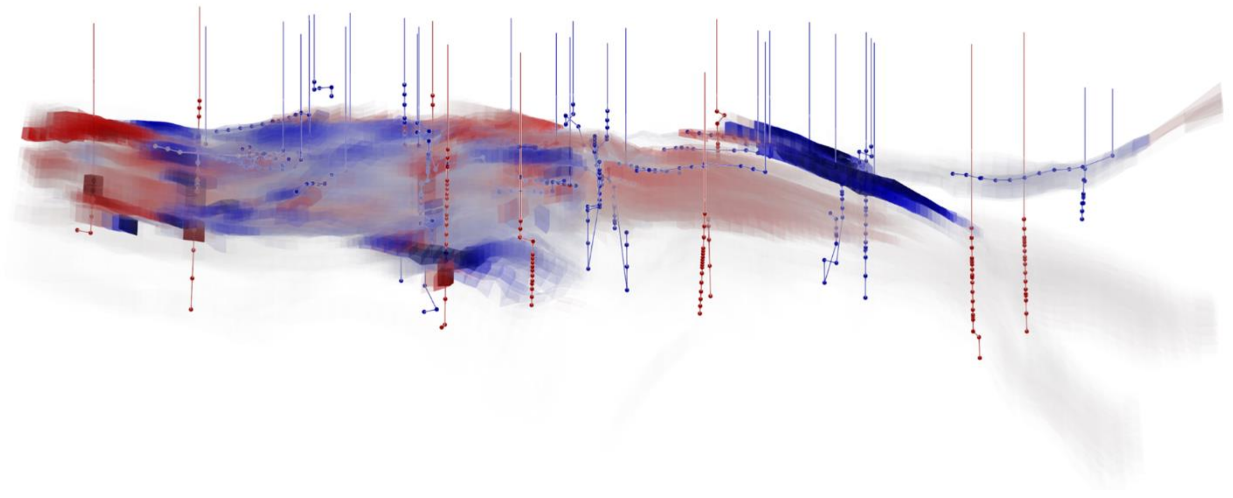
Gas Corey exponent

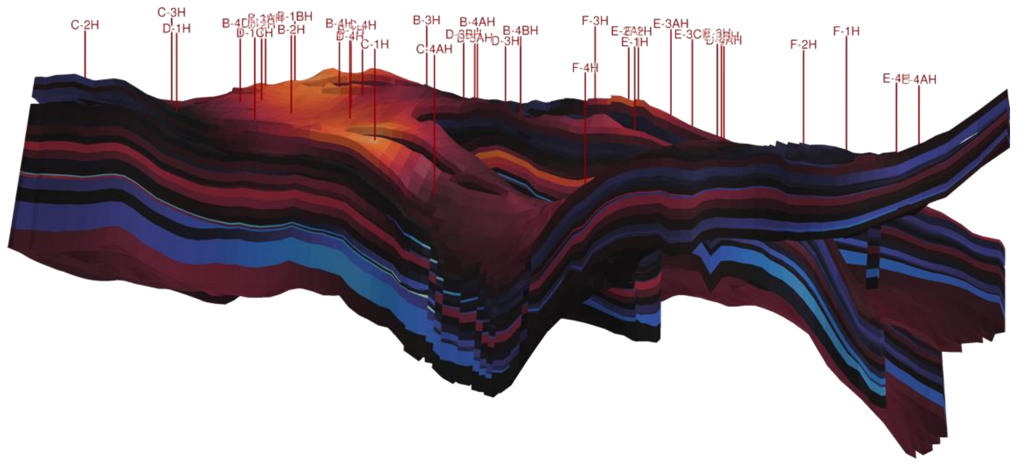


Liquid viscosity

Gradients: Norne gas production

```julia
total_time = sum(case.dt) # Normalize objective by total time
import JutulDarcy: compute_well_qoi, reservoir_sensitivities
function objective(model, state, Δt, step_i, forces)
    grat = 0.0
    T = SurfaceGasRateTarget # Get out gas rate
    for w in prod
        grat += compute_well_qoi(model, state, forces, w, T)
    end
    return Δt*grat/total_time # Total gas production, normalized by time
end
```





134k variables: 1 million numerical parameters

Parameter	Count	Type	Note
Transmissibilities	132693×1	Faces	Discretizes potential difference
Gravity difference $g\Delta z$	132693×1	Faces	Discretizes bouyancy
Connate water	44417×1	Cells	Used in evaluation of k_r
Water k_r scalers	44417×4	Cells	End-points and maximum k_r
Oil-water k_r scalers	44417×4	Cells	–
Oil-gas k_r scalers	44417×4	Cells	–
Gas k_r scalers	44417×4	Cells	–
Pore-volume	44417×1	Cells	Pore space available to flow
Well indices	503×1	Perforations	Connection well and reservoir
Perforation Δz	503×1	Perforations	Depth from well to perforation
Top node volume	36×1	Well	Volume for standard well

SINTEF

Input variables for Norne

Parameter	Count	Type	Impact
Areas	132693×1	Faces	Transmissibilities
Face centroids	132693×3	Faces	Transmissibilities
Normals	132693×3	Faces	Transmissibilities
Transmissibility multiplier	132693×1	Faces	Transmissibilities
Permeability	44417×3	Cells	Transmissibilities and well indices
Porosity	44417×1	Cells	Pore-volume
Net-to-gross	44417×1	Cells	Pore-volume, transmissibilities, well indices
Water k_r scalers	44417×4	Cells	Direct copy
Oil-water k_r scalers	44417×4	Cells	Direct copy
Oil-gas k_r scalers	44417×4	Cells	Direct copy
Gas k_r scalers	44417×4	Cells	Direct copy
Volumes	44417×1	Cells	Pore-volumes
Cell centroids	44417×3	Cells	$g\nabla z$, well perforation depth difference and transmissibilities

SINTEF

Concluding remarks

- Automatic differentiation (AD) has a reputation as slow
  - Careful application can give excellent performance and unparalleled flexibility
- Adjoints, fast AD and a systematic approach to parameters allows immediate understanding of what parameters impact a given objective function
- Ideas are not Julia specific - can easily be adapted to other codes
- Julia largely delivers on the promise of both scripting flexibility and performance
  - Some issues remain, like poor debugging support and latency when recompilation is required
  - Performance is excellent and multiple dispatch is very powerful
  - Package manager is great – and a good answer to build system despair and dependency hell

```
using Pkg; Pkg.add(["JutulDarcy", "MPI", "HYPRE", "GLMakie"])
```

SINTEF

Thank you for your attention!

https://github.com/sintefmath/JutulDarcy.jl