JutulDarcy.jl Documentation

Olav Møyner and contributors

2026-02-16

Contents

Chapter 1

JutulDarcy.jl

Figure 1.1: JutulDarcy Logo

Re-thinking reservoir simulation in Julia

High-performance porous media and reservoir simulator based on automatic differentiation

1.1 What is this?

JutulDarcy.jl is a general-purpose high-performance porous media simulator toolbox (CO2 sequestration, gas/H2 storage, oil/gas fields) written in Julia based on Jutul.jl, developed by the Applied Computational Science group at SINTEF Digital.

A few highlights:

- Immiscible, black-oil, compositional, CO2-brine and geothermal systems
- Fully differentiable through adjoint method (history matching of parameters, optimization of well controls)
- High performance, with optional support for compiling MPI parallel binaries
- Consistent discretizations
- Industry standard input formats - or make your own model as a script
- 3D visualization and tools for post-processing of simulation results

1.2 Key Features

1.2.1 Physical systems

Immiscible, compositional, geothermal and black-oil flow

1.2.2 Differentiability

Compute sensitivities of parameters with high-performance adjoint method

1.2.3 High performance on CPU & GPU

Fast execution with support for MPI, CUDA and thread parallelism

1.3 Quick start guide

Getting started is the main setup guide that includes the basics of installing Julia and creating a Julia environment for `JutulDarcy.jl`, written for users who may not already be familiar with Julia package management.

If you want to get started quickly: Install Julia and add the following packages together with a Makie backend library to your environment of choice using Julia's package manager `Pkg`:

```
using Pkg
Pkg.add("GLMakie")     # Plotting
Pkg.add("Jutul")       # Base package
Pkg.add("JutulDarcy") # Reservoir simulator
```

To verify that everything is working, we have a minimal example that runs an industry standard input file and produces interactive plots. Note that interactive plotting requires `GLMakie`, which may not work if you are running Julia over SSH.

1.3.1 Python bindings

Alternatively, the code has a Python package that can be installed using `pip`:

```
pip install jutuldarcy
```

1.3.2 Examples

The examples are published in the documentation. For a list of examples categorized by tags, see the Example overview page.

To get access to all the examples as code, you can generate a folder that contains the examples locally, you can run the following code to create a folder `jutuldarcy_examples` in your current working directory:

```
using JutulDarcy
generate_jutuldarcy_examples()
```

These examples can then be run using `include("jutuldarcy_examples/example_name.jl")` or opened in an editor to be run line by line.

1.4 Citing JutulDarcy

If you use JutulDarcy for a scientific publication, please cite the main paper the following way:

> O. Møyner, (2024). JutulDarcy.jl - a Fully Differentiable High-Performance Reservoir Simulator Based on Automatic Differentiation. Computational Geosciences (2025), Open Access: https://doi.org/10.1007/s10596-025-10366-6

BibTeX:

```
@article{jutuldarcy,
  title={JutulDarcy.jl - a fully differentiable high-performance reservoir simulator based on a
  author={M{\o}yner, Olav},
  journal={Computational Geosciences},
  volume={29},
  number={30},
  year={2025},
  publisher={Springer}
}
```

1.5 A few of the packages used by JutulDarcy

JutulDarcy.jl builds upon many of the excellent packages in the Julia ecosystem. Here are a few of them, and what they are used for:

- ForwardDiff.jl implements the Dual number class used throughout the code
- SparsityTracing.jl provides sparsity detection inside Jutul
- Krylov.jl provides the iterative linear solvers
- ILUZero.jl for ILU(0) preconditioners
- AlgebraicMultigrid.jl for AMG preconditioners
- HYPRE.jl for robust AMG preconditioners with MPI support
- PartitionedArrays.jl for MPI assembly and linear solve
- CUDA.jl for CUDA-GPU support
- AMGX.jl for AMG on CUDA GPUs
- Tullio.jl for automatically optimized loops and Polyester.jl for lightweight threads
- TimerOutputs.jl and ProgressMeter.jl gives nice output to terminal.
- Makie.jl is used for the visualization features
- MultiComponentFlash.jl provides many of the compositional features

…and many more directly, and indirectly - see the Project.toml and Manifest files for a full list! # Getting started

1.6 Installing JutulDarcy

To get started with JutulDarcy, you must install the Julia programming language. We recommend the latest stable release, but at least version 1.9 is required. Julia uses environments to manage packages. If you are not familiar with this concept, we recommend the Pkg documentation.

1.6.1 Setting up an environment

To set up an environment you can create a folder and open Julia with that project as a runtime argument. The environment is persistent: If you start Julia in the same folder with the same project argument, the same packages will be installed. For this reason, it is sufficient to do this process once.

```
mkdir jutuldarcy_env
cd jutuldarcy_env
julia --project=.
```

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

You can then run any of the examples in the `examples` directory by including them. The examples are sorted by complexity. We suggest you start with Your first JutulDarcy.jl simulation.

To generate a folder that contains the examples locally, you can run the following code to create a folder `jutuldarcy_examples` in your current working directory:

```
using JutulDarcy
generate_jutuldarcy_examples()
```

Alternatively, a folder can be specified if you want the examples to be placed outside your present working directory:

```
using JutulDarcy
generate_jutuldarcy_examples("/home/username/")
```

1.6.2 Adding additional packages

We also rely heavily on the Jutul base package for some functionality, so we recommend that you also install it together with JutulDarcy:

```
Pkg.add("Jutul")
```

If you want the plotting used in the examples, you need this:

```
Pkg.add("GLMakie") # 3D and interactive visualization
```

Some examples and functionalites also make use of additional packages:

```
Pkg.add("Optim") # Optimization library
Pkg.add("HYPRE") # Better linear solver
Pkg.add("GeoEnergyIO") # Parsing input files
```

Chapter 2

Your first JutulDarcy.jl simulation

After a bit of time has passed compiling the packages, you are now ready to use JutulDarcy. There are a number of examples included in this manual, but we include a brief example here that briefly demonstrates the key concepts.

2.1 Setting up the domain

We set up a simple Cartesian Mesh that is converted into a reservoir domain with static properties permeability and porosity values together with geometry information. We then use this domain to set up two wells: One vertical well for injection and a single perforation producer.

2.2 Setting up a fluid system

We select a two-phase immiscible system by declaring that the liquid and vapor phases are present in the model. These are assumed to have densities of 1000 and 100 kilograms per meters cubed at reference pressure and temperature conditions.

2.3 Setting up the model

We now have everything we need to set up a model. We call the reservoir model setup function and get out the model together with the parameters. Parameter represent numerical input values that are static throughout the simulation. These are automatically computed from the domain's geometry, permeability and porosity.

The model has a set of default secondary variables (properties) that are used to compute the flow equations. We can have a look at the reservoir model to see what the defaults are for the Darcy flow part of the domain:

The secondary variables can be swapped out, replaced and new variables can be added with arbitrary functional dependencies thanks to Jutul's flexible setup for automatic differentiation. Let us adjust the defaults by replacing the relative permeabilities with Brooks-Corey functions and the phase density functions by constant compressibilities:

2.4 Initial state: Pressure and saturations

Now that we are happy with our model setup, we can designate an initial state. Equilibration of reservoirs can be a complicated affair, but here we set up a constant pressure reservoir filled with the liquid phase. The inputs must match the primary variables of the model, which in this case is pressure and saturations in every cell.

2.5 Setting up timesteps and well controls

We set up reporting timesteps. These are the intervals that the simulator gives out outputs. The simulator may use shorter steps internally, but will always hit these points in the output. Here, we report every year for 25 years.

We next set up a rate target with a high amount of gas injected into the model. This is not fully realistic, but will give some nice and dramatic plots for our example later on.

The producer is set to operate at a fixed pressure:

We can finally set up forces for the model. Note that while JutulDarcy supports time-dependent forces and limits for the wells, we keep this example as simple as possible.

2.6 Simulate and analyze results

We call the simulation with our initial state, our model, the timesteps, the forces and the parameters:

We can interactively look at the well results in the command line:

Let us look at the pressure evolution in the injector:

If we have a plotting package available, we can visualize the results too:

Interactive visualization of the 3D results is also possible if GLMakie is loaded:

Chapter 3

Frequently asked questions

Here are a few common questions and possible answers. You may also want to have a look at the GitHub issues and the GitHub discussions page.

3.1 Input and output

3.1.1 What input formats can JutulDarcy.jl use?

1. DATA files (used by Eclipse, OPM Flow, tNavigator, Echelon and others) provided that the grid is given either as a corner-point GRDECL file or in TOPS format. As with most reservoir simulators, not all features of the original format are supported, but the code will let you know when unsupported features are encountered.
2. Cases written out from MRST through the `jutul` module.
3. Cases written entirely in Julia using the basic `Jutul` and `JutulDarcy` data structures, as seen in the examples of the module.

3.1.2 What output formats does JutulDarcy.jl have?

The simulator outputs results into standard Julia data structures (e.g. Vectors and Dicts) that can easily be written out using other Julia packages, for example in CSV format. We do not currently support binary formats output by commercial simulators.

Simulation results are written to disk using JLD2, a subset of HDF5 commonly used in Julia for storing objects to disk.

3.1.3 How do I restart an interrupted simulation?

JutulDarcy keeps everything in memory by default. This is not practical for larger models. If the argument `output_path` is set to a directory, JutulDarcy writes to the `JLD2` format (variant of HDF5).

```
# Note: set ENV["JUTUL_OUTPUT_PATH"] in your startup.jl first!
pth = jutul_output_path("My_test_case")
simulate_reservoir(case, output_path = pth)
```

If an output path is set, you can restart simulations:

```
# Restart from the last successfully solved step, or return output if everything is simulated
ws, states = simulate_reservoir(case, output_path = pth, restart = true)
# Start from the beginning, overwriting files if already present
ws, states = simulate_reservoir(case, output_path = pth, restart = false)
# Restart from step 10 and throw error if step 9 is not already stored on disk.
ws, states = simulate_reservoir(case, output_path = pth, restart = 10)
```

You can restart the simulation with different options for timestepping or tolerances.

3.1.4 How do I decide where output is stored?

`Jutul.jl` contains a system for managing output folders. It is highly recommended that you amend your `startup.jl` file to include `ENV["JUTUL_OUTPUT_PATH"]` that points to where you want output to be stored. For example, on Windows usage of the output path mechanism may look something like this:

```
julia> ENV["JUTUL_OUTPUT_PATH"]
"D:/jutul_output/"

julia> jutul_output_path() # Randomly generated file name
"D:/jutul_output/jutul/jl_DwpAvQTiLo"

julia> jutul_output_path("mycase")
"D:/jutul_output/jutul/mycase"

julia> jutul_output_path("mycase", subfolder = "ensemble_name")
"D:/jutul_output/ensemble_name/mycase"

julia> jutul_output_path("mycase", subfolder = missing)
"D:/jutul_output/mycase"
```

Or equivalent on a Linux system:

```
julia> ENV["JUTUL_OUTPUT_PATH"]
"/home/username/jutul_output/"

julia> jutul_output_path() # Randomly generated file name
"/home/username/jutul_output/jutul/jl_DwpAvQTiLo"

julia> jutul_output_path("mycase")
"/home/username/jutul_output/jutul/mycase"

julia> jutul_output_path("mycase", subfolder = "ensemble_name")
"/home/username/jutul_output/ensemble_name/mycase"

julia> jutul_output_path("mycase", subfolder = missing)
"/home/username/jutul_output/mycase"
```

You can also just specify a full path and keep track of output folders yourself, but using the

`jutul_output_path` mechanism will make it easier to write a script that can be run on another computer with different folder structure.

3.1.5 How do you get out more output from a simulation?

The default outputs per cell are primary variables and total masses:

```
reservoir_model(model).output_variables
3-element Vector{Symbol}:
 :Pressure
 :Saturations
 :TotalMasses
```

You can push variables to this list, or ask the code to output all variables:

```
model2, = setup_reservoir_model(domain, sys, extra_outputs = true);
reservoir_model(model2).output_variables
7-element Vector{Symbol}:
 :Pressure
 :Saturations
 :TotalMasses
 :PhaseMassDensities
 :RelativePermeabilities
 :PhaseMobilities
 :PhaseMassMobilities
```

You can also pass `extra_outputs = [:PhaseMobilities]` as a keyword argument to `setup_reservoir_model` to make the resulting model output specific variables.

3.1.6 What is the unit and sign convention for well rates?

Well results are given in strict SI, which means that rates are generally given in m^3/s. Rates are positive for injection (mass entering the reservoir domain) and negative for production (leaving the reservoir domain).

3.2 Plotting

3.2.1 What is required for visualization?

We use the wonderful `Makie.jl` for both 2D and 3D plots. Generally `CairoMakie` is supported for non-interactive plotting and `GLMakie` is used for interactive plotting (especially 3D). The latter requires a working graphics context, which is not directly available when the code is run over for example SSH or on a server.

For more details on the backends, see the`Makie.jl` docs

3.3 Miscellaneous

3.3.1 Can you add feature X or format Y?

If you have a feature you'd like to have supported, please file an issue with details on the format. `JutulDarcy` is developed primarily through contract research, so features are added as needed for ongoing projects where the simulator is in use. Posting an issue, especially if you have a clear reference to how something should be implemented is still very useful. It is also possible to fund the development for a specific feature, or to implement the feature yourself by asking for pointers on how to get started.

3.3.2 What units does JutulDarcy.jl use?

JutulDarcy uses consistent units. This typically means that all your values must be input in strict SI. This means pressures in Pascal, temperatures in Kelvin and time in seconds. Note that this is very similar to the `METRIC` type of unit system seen in many commercial simulators, except that units of time is not given in days. This also impacts permeabilities, transmissibilities and viscosities, which will be much smaller than in metric where days are used.

Reading of input files will automatically convert data to the correct units for simulation, but care must be taken when you are writing your own code. `Jutul.jl` contains unit conversion factors to make it easier to write code:

You can also extract individual units and to the setup yourself:

JutulDarcy does currently not make use of conversion factors or explicit units can in principle use any consistent unit system. Some default scaling of variables assume that the magnitude pressures and velocities roughly match that of strict SI (e.g. Pascals and cubic meters per second). These scaling factors are primarily used when iterative linear solvers are used.

3.3.3 Who develops JutulDarcy.jl?

The module is developed and maintained by the Applied Computational Sciences group at SINTEF Digital. SINTEF is one of Europe's largest independent research organizations and is organized as a not-for-profit institute. Olav Møyner is the primary maintainer.

3.3.4 Why write a new reservoir simulation code in Julia?

We believe that reservoir simulation should be a *library* and not necessarily an application by itself. The future of porous media simulation is deeply integrated into other workflows, and not as an application that simply writes files to disk. As a part of experimentation in differentiable and flexible solvers using automatic differentiation that started with MRST, Julia was the natural next step.

3.3.5 What is the license of JutulDarcy.jl?

The code uses the MIT license, a permissive license that requires attribution, but does not place limitations on commercial use or closed-source integration.

The code uses a number of dependencies that can have other licenses and we make no guarantees that the entirety of the code made available by adding `JutulDarcy.jl` to a given Julia environment is all MIT licensed. # High-level API

3.4 Setup

The basic outline of building a reservoir simulation problem consists of:

1. Making a mesh
2. Converting the mesh into a reservoir, adding properties
3. Add any number of wells
4. Setup a physical system and setup a reservoir model
5. Set up timesteps, well controls and other forces
6. Simulate!

3.4.1 Meshes

JutulDarcy can use meshes that are supported by Jutul. This includes the Cartesian (Jutul.CartesianMesh) and Unstructured meshes (Jutul.UnstructuredMesh), meshes from Gmsh (Jutul.mesh_from_gmsh), meshes from MRST (Jutul.MRSTWrapMesh), and meshes from the Meshes.jl package.

3.4.2 Reservoir

Once a mesh has been set up, we can turn it into a reservoir with static properties:

3.4.3 Wells

Wells are most easily created using utilities that act directly on a reservoir domain:

3.4.4 Model

A single, option-heavy function is used to set up the reservoir model and default parameters:

3.4.5 Initial state

The initial state can be set up by explicitly setting all primary variables. JutulDarcy also contains functionality for initial hydrostatic equilibriation of the state, which is either done by setting up `EquilibriumRegion` instances that are passed to `setup_reservoir_state`, or by using an input file with the `EQUIL` keyword.

3.5 Simulation

Simulating is done by either setting up a reservoir simulator and then simulating, or by using the convenience function that automatically sets up a simulator for you.

There are a number of different options available to tweak the tolerances, timestepping and behavior of the simulation. It is advised to read through the documentation in detail before running very large simulations.

Chapter 4

Input formats

It is also possible to read cases that have been set up in MRST (see setup_case_from_mrst and simulate_mrst_case) or from .DATA files (see setup_case_from_data_file and simulate_data_file)

4.1 MAT-files from the Matlab Reservoir Simulation Toolbox (MRST)

4.1.1 Simulation of .MAT files

4.1.2 MRST-specific types

4.2 DATA-files from commercial reservoir modelling software

JutulDarcy can set up cases from Eclipse and Intersect type input files by making use of the GeoEnergyIO.jl package for parsing. This package is a direct dependency of JutulDarcy and these cases can be simulated directly. If you want to parse the input files and possibly modify them in your Julia session before the case is simulated, we refer you to the GeoEnergyIO Documentation.

If you want to directly simulate a file from disk, you can sue the high level functions that automatically parse the files for you:

Reservoir simulator input files are highly complex and contain a great number of different keywords. JutulDarcy and GeoEnergyIO has extensive support for this format, but many keywords are missing or only partially supported. Sometimes cases can be made to run by removing keywords that do not impact simulation outcomes, or have very little impact. If you want a turnkey open source solution for simulation reservoir models you should also have a look at OPM Flow.

If you can share your input file or the missing keywords in the issues section it may be easier to figure out if a case can be supported.

4.3 Premade models

4.3.1 SPE10, model 2

Chapter 5

Supported physical systems

JutulDarcy supports a number of different systems. These are `JutulSystem` instances that describe a particular type of physics for porous media flow. We describe these in roughly the order of complexity that they can model.

5.1 Summary

The general form of the flow systems we will discuss is a conservation law for N components on residual form:

$$R = \frac{\partial}{\partial t} M_i + \nabla \cdot \vec{V}_i - Q_i, \quad \forall i \in \{1, ..., N\}$$

Here, M_i is the conserved quantity (usually masses) for component i and $\vec{V}_i$ the velocity of the conserved quantity. Q_i represents source terms that come from direct sources SourceTerm, boundary conditions (FlowBoundaryCondition) or from wells (setup_well, setup_vertical_well).

The following table gives an overview of the available features that are described in more detail below:

System	Number of phases	Number of components	M	V
SinglePhaseSystem	1	1	$\rho\phi$	$\rho\vec{v}$
ImmiscibleSystem	Any	(Any)	$S_\alpha\rho_\alpha\phi$	$\rho_\alpha\vec{v}_\alpha$
StandardBlackOilSystem	2-3	(2-3)	$\rho_o^s(b_g S_g + R_s b_o S_o)$	$b_g\vec{v}_g + R_s b_g\vec{v}_o$
MultiPhaseCompositionalSystemLV	2-3	Any	$\rho_l X_i S_l + \rho_v Y_i S_v$	$\rho_l X_i\vec{v}_l + \rho_v Y_i\vec{v}_v$

5.1.1 Phases

Phases are defined using specific types. Some constructors take a list of phases present in the model. Phases do not contain any data themselves and the distinction between different phases applies primarily for well controls.

The phases are used by subtypes of the abstract superclass for multiphase flow systems:

5.1.2 Implementation details

In the above the discrete version of M_i is implemented in the update function for JutulDarcy.TotalMasses that should by convention be named JutulDarcy.update_total_masses!. The discrete component fluxes are implemented by JutulDarcy.component_mass_fluxes!.

The source terms are implemented by Jutul.apply_forces_to_equation! for boundary conditions and sources, and Jutul.update_cross_term_in_entity! for wells. We use Julia's multiple dispatch to pair the right implementation with the right physics system.

5.2 Single-phase flow

The simplest form of porous media flow is the single-phase system.

$$r(p) = \frac{\partial}{\partial t}(\rho\phi) + \nabla \cdot (\rho\vec{v}) - \rho q$$

ρ is the phase mass density and ϕ the apparent porosity of the medium, i.e. the void space in the rock available to flow. Where the velocity $\vec{v}$ is given by Darcy's law that relates to the pressure gradient ∇p and hydrostatic head to the velocity field:

$$\vec{v} = -\frac{\mathbf{K}}{\mu}(\nabla p + \rho g \nabla z)$$

Here, $\mathbf{K}$ is a positive-definite permeability tensor, μ the fluid viscosity, g the magnitude of gravity oriented down and z the depth.

!!! note "Single-phase implementation" The SinglePhaseSystem is a dedicated single phase system. This is mathematically equivalent to an ImmiscibleSystem when set up with a single phase. For single phase flow, the fluid Pressure is the primary variable in each cell. The equation supports two types of compressibility: That of the fluid where density is a function $\rho(p)$ of pressure and that of the pores where the porosity $\phi(p)$ changes with pressure.

!!! tip JutulDarcy uses the notion of depth rather than coordinate when defining buoyancy forces. This is consistent with the convention in the literature on subsurface flow.

5.3 Multi-phase, immiscible flow

The flow systems immediately become more interesting if we add more phases. We can extend the above single-phase system by introducing the phase saturation of phase with label α as S_α. The phase saturation represents the volumetric fraction of the rock void space occupied by the phase. If we consider a pair of phases $\{n, w\}$ non-wetting and wetting we can write the system as

$$r_\alpha = \frac{\partial}{\partial t}(S_\alpha \rho_\alpha \phi) + \nabla \cdot (\rho_\alpha \vec{v}_\alpha) - \rho_\alpha q_\alpha = 0, \quad \alpha \in \{n, w\}$$

This requires an additional closure such that the amount of saturation of all phases exactly fills the available fluid volume:

$$S_w + S_n = 1, \quad 1 \geq S_\alpha \geq 0 \quad \alpha \in \{n, w\}$$

This equation is local and linear in the saturations and can be eliminated to produce the classical two-equation system for two-phase flow,

$$r_n = \frac{\partial}{\partial t}((1 - S_w)\rho_n\phi) + \nabla \cdot (\rho_n \vec{v}_n) - \rho_n q_n = 0, r_w = \frac{\partial}{\partial t}(S_w\rho_w\phi) + \nabla \cdot (\rho_w \vec{v}_w) - \rho_w q_w = 0.$$

To complete this description we also need expressions for the phase fluxes. We use the standard multiphase extension of Darcy's law,

$$\vec{v}_\alpha = -\mathbf{K}\frac{k_{r\alpha}}{\mu_\alpha}(\nabla p\alpha + \rho_\alpha g \nabla z)$$

Here, we have introduced the relative permeability of the phase $k_{r\alpha}$, an empirical relationship between the saturation and the flow rate. Relative permeability is a complex topic with many different relationships and functional forms, but we limit the discussion to monotone, non-negative functions of their respective saturations, for example a simple Brooks-Corey type of $k_{r\alpha}(S_\alpha) = S_\alpha^2$. We have also introduced separate phase pressures p_α that account for capillary pressure, e.g. $p_w = p_n + p_c(S_w)$.

5.3.1 Primary variables

!!! note "Immiscible implementation" The ImmiscibleSystem implements this system for any number of phases. The primary variables for this system is a single reference Pressure and phase Saturations. As we do not solve for the volume closure equation, there is one less degree of freedom associated with the saturations than there are number of phases.

5.4 Black-oil: Multi-phase, pseudo-compositional flow

The black-oil equations is an extension of the immiscible description to handle limited miscibility between the phases. Originally developed for certain types of oil and gas simulation, these equations are useful when the number of components is low and tabulated values for dissolution and vaporization are available.

The assumptions of the black-oil model is that the "oil" and "gas" pseudo-components have uniform composition throughout the domain. JutulDarcy supports two- and three-phase black oil flow. The difference between two and three phases amounts to an additional immiscible aqueous phase that is identical to that of the previous section. For that reason, we focus on the miscible pseudo-components, oil:

$$r_o = \rho_o^s \left(\frac{\partial}{\partial t}((b_o S_o + R_v b_g(1 - S_o))\phi) + \nabla \cdot (b_o \vec{v}_o + R_v b_o \vec{v}_g) - q_o^s \right),$$

and gas,

$$r_g = \rho_g^s \left(\frac{\partial}{\partial t}((b_g S_g + R_s b_o S_o)\phi) + \nabla \cdot (b_g \vec{v}_g + R_s b_g \vec{v}_o) - q_g^s \right)$$

The model uses the notion of surface (or reference densities) ρ_o^s, ρ_g^s to define the densities of the component at specific pressure and temperature conditions where it is assumed that all "gas" has moved to the vapor phase and the defined "oil" is only found in the liquid phase. Keeping this definition in mind, the above equations can be divided by the surface densities to produce a surface volume balance equation where we have defined `b_o` and `b_g` as the dimensionless reciprocal formation volume factors that relate a volume at reservoir conditions to surface volumes and `R_s` for the dissolved volume of gas in the oil phase when brought to surface conditions. `R_v` is the same definition, but for oil vaporized into the gas phase.

!!! note "Blackoil implementation" The StandardBlackOilSystem implements the black-oil equations. It is possible to run cases with and without water, with and without R_s and with and without R_v. The primary variables for the most general case is the reference Pressure, an ImmiscibleSaturation for the aqueous phase and the special BlackOilUnknown that will represent either S_o, R_s or R_v on a cell-by-cell basis depending on what phases are present and saturated.

A full description of the black-oil equations is outside the scope of this documentation. Please see mrst-book-i for more details.

5.5 Compositional: Multi-phase, multi-component flow

The more general case of multi-component flow is often referred to as a compositional model. The typical version of this model describes the fluid as a system of N components where the phases present and fluid properties are determined by an equation-of-state. This can be highly accurate if the equation-of-state is tuned for the mixtures that are encountered, but comes at a significant computational cost as the equation-of-state must be evaluated many times.

JutulDarcy implements a standard compositional model that assumes local instantaneous equilibrium and that the components are present in up to two phases with an optional immiscible phase added. This is sometimes referred to as a "simple water" or "dead water" description. By default the solvers use MultiComponentFlash.jl to solve thermodynamic equilibrium. This package implements the generalized cubic approach and defaults to Peng-Robinson.

Assume that we have two phases liquid and vapor referred to as l and v with the Darcy flux given as in the preceeding sections. We can then write the residual equation for each of the M components by the liquid and vapor mole fractions X_i, Y_i'of that component as:

$$r_i = \frac{\partial}{\partial t}\left((\rho_l X_i S_l + \rho_v Y_i S_v)\phi\right) + \nabla \cdot (\rho_l X_i \vec{v}_l + \rho_v Y_i \vec{v}_v) - Q_i, \quad M \in \{1, \dots, M\}$$

For additional details, please see Chapter 8 - Compositional Simulation with the AD-OO Framework in mrst-book-ii.

!!! note "Compositional implementation" The MultiPhaseCompositionalSystemLV implements the compositional model. The primary variables for the most general case is the reference Pressure, an ImmiscibleSaturation for the optional immiscible phase and $M - 1$ OverallMoleFractions.

5.6 Multi-phase thermal flow

Thermal effects are modelled as an additional residual equation that comes in addition to the flow equations.

$$r_i = \frac{\partial}{\partial t}\Big(\rho_r U_r(1-\phi) + \sum_\alpha \rho_\alpha S_\alpha U_\alpha \phi\Big) + \nabla \cdot \left(\sum_\alpha (H_\alpha \rho_\alpha v_\alpha - S_\alpha \lambda_\alpha \nabla T) - \lambda_r \nabla T\right) - Q_e$$

Chapter 6

Solving the equations

By default, Jutul solves a system as a fully-coupled implicit system of equations discretized with a two-point flux approximation with single-point upwind.

6.1 Newton's method

The standard way of solving a system of non-linear equations is by Newton's method (also known as Newton-Raphson's method). A quick recap: For a vector valued residual $\mathbf{r}(x)$ of the primary variable vector $\mathbf{x}$ we can defined a Newton update:

$$\mathbf{x}^{k+1} = \mathbf{r}^k - J^{-1}\mathbf{r}(\mathbf{x}^k), \quad J_{ij} = \frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_j}.$$

JutulDarcy solves systems that generally have both non-smooth behavior and physical constraints on the values for $\mathbf{x}$. For that reason, we modify Newton's method slightly:

$$\mathbf{x}^{k+1} = \mathbf{r}^k + \omega(\Delta \mathbf{x})$$

Here, ω is a function that limits the variables so that they do not change too much (e.g. Appleyard chopping, limiting of pressure, saturation and composition updates) and that they are within the prescribed limits. There are also options for automated global dampening in the presence of convergence issues. The update is then defined from inverting the Jacobian:

$$\Delta \mathbf{x} = -J^{-1}\mathbf{r}(\mathbf{x}^k), \quad J_{ij} = \frac{\partial \mathbf{r}_i}{\partial \mathbf{x}_j}.$$

Starting with $\mathbf{x}^0$as some initial guess taken from the previous time-step, we can solve the system by iterating upon this loop.

6.2 Linear solvers and linear systems

For most practical applications it is not feasible or efficient to invert the Jacobian. JutulDarcy uses preconditioned iterative solvers by default, but it is possible to use direct solvers as well

when working with smaller models. The high level interface for setting up a reservoir model setup_reservoir_model has an optional `block_backend=true` keyword argument that determines the matrix format, and consequently the linear solver type to be used.

6.2.1 Direct solvers

If `block_backend` is set to `false`, Jutul will assemble into the standard Julia CSC sparse matrix with `Float64` elements and Julia's default direct solver will be used. It is also possible to use other Julia solvers on this system, but the default preconditioners assume that block backend is enabled.

6.2.2 Iterative solver

If `block_backend` is set to `true`, Jutul will by default use a constrained-pressure residual (CPR) preconditioner for BiCGStab. Jutul relies on Krylov.jl for iterative solvers. The main function that selects the linear solver is reservoir_linsolve that allows for the selection of different preconditioners and linear solvers. This is often an instance of Jutul.GenericKrylov with the approprioate preconditioner.

6.2.2.1 Single model (only porous medium)

If the model is a single model (e.g. only a reservoir) the matrix format is a block-CSC matrix that combines Julia's builtin sparse matrix format with statically sized elements from the StaticArrays.jl package. If we consider the two-phase immiscible system from Multi-phase, immiscible flow we have a pair of equations R_n, R_w together with the corresponding primary variables pressure and first saturation p, S_n defined for all N_c cells. Let us simplify the notation a bit so that the subscripts of the primary variables are p, s and define a $N_c \times N_c$ block Jacobian linear system where the entires are given by:

$$J_{ij} = \begin{bmatrix} \left(\frac{\partial r_n}{\partial p}\right)_{ij} & \left(\frac{\partial r_n}{\partial s}\right)_{ij} \\ \left(\frac{\partial r_w}{\partial p}\right)_{ij} & \left(\frac{\partial r_w}{\partial s}\right)_{ij} \end{bmatrix} = \begin{bmatrix} J_{np} & J_{ns} \\ J_{wp} & J_{ws} \end{bmatrix}_{ij}$$

This block system has several advantages:

- We immediately get access to more powerful version of standard Julia preconditioners provided that all operations used are applicable for matrices and are applied in the right commutative order. For example, JutulDarcy uses the ILUZero.jl package when a CSC linear system is preconditioned with incomplete LU factorization with zero fill-in.
- Sparse matrix vector products are much more efficient as less indicies need to be looked up for each element wise multiplication.
- Performing local reductions over variables is much easier when they are located in a local matrix.

6.2.2.1.1 Constrained Pressure Residual The CPR preconditioner wallis-cpr, cao-cpr CPRPreconditioner is a multi-stage physics-informed preconditioner that seeks to decouple the global pressure part of the system from the local transport part. In the limits of incompressible flow without gravity it can be thought of as an elliptic / hyperbolic splitting. We also implement a special variant for the adjoint system that is similar to the treatment described in adjoint_cpr.

The short version of the CPR preconditioner can be motivated by our test system:

$$r_n = \frac{\partial}{\partial t}((1 - S_w)\rho_n\phi) + \nabla \cdot (\rho_n \vec{v}_n) - \rho_n q_n = 0, r_w = \frac{\partial}{\partial t}(S_w\rho_w\phi) + \nabla \cdot (\rho_w \vec{v}_w) - \rho_w q_w = 0.$$

For simplicity, we assume that there is no gravity, source terms, or compressibility. Each equation can then be divided by their respective densities and summed up to produce a pressure equation:

$$r_p = \frac{\partial}{\partial t}((1-S_w)\phi)+\nabla\cdot\vec{v}_n+\frac{\partial}{\partial t}(S_w\phi)+\nabla\cdot\vec{v}_w = \frac{\partial}{\partial t}((S_w-S_w)\phi)+\nabla\cdot(\vec{v}_n+\vec{v}_w) = \nabla\cdot(\vec{v}_n+\vec{v}_w) = -\nabla\mathbf{K}(k_{rw}/\mu_w+k_{rn}/\mu_n$$

The final equation is the variable coefficient Poisson equation and is referred to as the incompressible pressure equation for a porous media. We know that algebraic multigrid preconditioners (AMG) are highly efficient for linear systems made by discretizing this equation. The idea in CPR is to exploit this by constructing an approximate pressure equation that is suited for AMG inside the preconditioner.

Constructing the preconditioner is done in two stages:

1. First, weights for each equation is found locally in each cell that decouples the time derivative from the non-pressure variables. In the above example, this was the true IMPES weights (dividing by density). JutulDarcy supports analytical true IMPES weights for some systems, numerical true IMPES weights for all systems and quasi IMPES weights for all systems.
2. A pressure equation is formed by weighting each equation by the respective weights and summing. We then have two systems: The pressure system r_p with scalar entries and the full system r that has block structure.

During the linear solve, the preconditioner is then made up of two broad stages: First, a preconditioner is applied to the pressure part (typically AMG), then the full system is preconditioned (typically ILU(0)) after the residual has been corrected by the pressure estimate:

1. Form weighted pressure residual $r_p = \sum_i w_i r_i$.
2. Apply pressure preconditioer M_p: $\Delta p = M_p^{-1} r_p$.
3. Correct global residual $r^* = r - JP(\Delta p)$ where P expands the pressure update to the full system vector, with zero entries outside the pressure indices.
4. Precondition the full system $\Delta x^* = M^{-1} r^*$
5. Correct the global update with the pressure to obtain the final update: $\Delta x = \Delta x^* + P(\Delta p)$

6.2.2.2 Multi model (porous medium with wells)

If a model is a porous medium with wells, the same preconditioners can be used, but an additional step is required to incorporate the well system. In practical terms, this means that our linearized system is expanded to multiple linear systems:

$$J\Delta\mathbf{x} = \begin{bmatrix} J_{rr} & J_{rw} \\ J_{wr} & J_{ww} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_r \\ \Delta\mathbf{x}_w \end{bmatrix} = \begin{bmatrix} \mathbf{r}_r \\ \mathbf{r}_w \end{bmatrix}$$

Here, J_{rr} is the reservoir equations differentiated with respect to the reservoir primary variables, i.e. the Jacobian from the previous section. J_{ww} is the well system differentiated with respect to

the well primary variables. The cross terms, $J_{rw}$and J_{wr}, are the same equations differentiated with respect to the primary variables of the other system.

The well system is generally much smaller than the reservoir system and can be solved by a direct solver. We would like to reuse the block preconditioners defined for the base system. The approach we use is a Schur complement approach to solve the full system. If we linearly eliminate the dependence of the reservoir equations on the well primary variables, we obtain the reduced system:

$$J\Delta\mathbf{x} = \begin{bmatrix} J_{rr} - J_{rw}J_{ww}^{-1}J_{wr} & 0 \\ J_{wr} & J_{ww} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{x}_r \\ \Delta\mathbf{x}_w \end{bmatrix} = \begin{bmatrix} \mathbf{r}_r - J_{rw}J_{ww}^{-1}\mathbf{r}_w \\ \mathbf{r}_w \end{bmatrix}$$

We can then solve the system in terms of the reservoir degrees of freedom where the system is a block linear system and we already have a working preconditioner:

$$\left(J_{rr} - J_{rw}J_{ww}^{-1}J_{wr}\right)\mathbf{x}_r = \mathbf{r}_r - J_{rw}J_{ww}^{-1}\mathbf{r}_w$$

Once that system is solved for $\mathbf{x}_r$, we can recover the well degrees of freedom $\mathbf{r}_w$ directly:

$$\mathbf{r}_w = J_{ww}^{-1}(\mathbf{r}_w - J_{wr}\mathbf{x}_r)$$

!!! note "Efficiency of Schur complement" Explicitly forming the matrix $J_{rr} - J_{rw}J_{ww}^{-1}J_{wr}$ will generally lead to a lot of fill-in in the linear system. JutulDarcy instead uses the action of $J_{rr} - J_{rw}J_{ww}^{-1}J_{wr}$ as a linear operator from LinearOperators.jl. This means that we must apply the inverse of the well system every time we need to compute the residual or action of the system matrix, but fortunately performing the action of the Schur complement is inexpensive as long as J_{ww} is small and the factorization can be stored. # Driving forces

6.3 Source terms

6.4 Boundary conditions

Chapter 7

Wells and controls

7.1 Well setup routines

Wells can be set up using the convenience functions setup_well and setup_vertical_well. These routines act on the output from reservoir_domain and can set up both types of wells. We recommend that you use these functions instead of manually calling the well constructors.

7.2 Types of wells

7.2.1 Simple wells

7.2.1.1 Equations

7.2.2 Multisegment wells

7.3 Well controls and limits

7.3.1 Types of well controls

7.3.2 Types of well targets

7.3.3 Implementation of well controls

7.3.4 Well outputs

7.3.5 Imposing limits on wells (multiple constraints)

7.4 Well forces

7.4.1 Perforations and WI adjustments

7.4.2 Other forces

Can use SourceTerm or FlowBoundaryCondition # Primary variables

7.5 Fluid systems

7.5.1 General

7.5.2 Immiscible flow

7.5.3 Black-oil flow

7.5.4 Compositional flow

7.6 Wells

7.7 WellGroup / Facility

Chapter 8

Secondary variables (properties)

8.1 Fluid systems

8.1.1 General

8.1.1.1 Relative permeabilities

The `ReservoirRelativePermeabilities` type also supports hysteresis for either phase.

8.1.1.2 Phase viscosities

8.1.1.3 Phase densities

8.1.1.4 Shrinkage factors

8.1.2 Black-oil flow

8.1.3 Compositional flow

8.2 Wells

Chapter 9

Parameters

9.1 General

9.2 Reservoir parameters

9.2.1 Transmissibility

9.2.2 Other

9.3 Well parameters

9.4 Thermal

Chapter 10

Plotting and visualization

Chapter 11

Utilities

This section describes various utilities that do not fit in other sections.

11.1 CO2 and brine correlations

These functions are not exported, but can be found inside the `CO2Properties` submodule. The functions described here are a Julia port of the MRST module described in salo_co2 They can be accessed by explicit import:

```
import JutulDarcy.CO2Properties: name_of_function
```

11.2 Relative permeability functions

11.3 CO2 inventory

11.4 API utilities

11.5 Model reduction

11.6 Adjoints and gradients

11.7 Well outputs

11.8 Non-neighboring connections

Chapter 12

Multi-threading and MPI support

JutulDarcy can use threads by default, but advanced options can improve performance significantly for larger models.

12.1 Overview of parallel support

There are two main ways of exploiting multiple cores in Jutul/JutulDarcy: Threads are automatically used for assembly and can be used for parts of the linear solve. If you require the best performance, you have to go to MPI where the linear solvers can use a parallel BoomerAMG preconditioner via HYPRE.jl. In addition, there is experimental GPU support described in GPU support.

12.1.1 MPI parallelization

MPI parallelizes all aspects of the solver using domain decomposition and allows a simulation to be divided between multiple nodes in e.g. a supercomputer. It is significantly more cumbersome to use than standard simulations as the program must be launched in MPI mode. This is typically a non-interactive process where you launch your MPI processes and once they complete the simulation the result is available on disk. The MPI parallel option uses a combination of MPI.jl, PartitionedArrays.jl and HYPRE.jl.

12.1.2 Thread parallelization

JutulDarcy also supports threads. By default, this only parallelizes property evaluations and assembly of the linear system. For many problems, the linear solve is the limiting factor for performance. Using threads is automatic if you start Julia with multiple threads.

An experimental thread-parallel backend for matrices and linear algebra can be enabled by setting `backend=:csr` in the call to setup_reservoir_model. This backend provides additional features such as a parallel zero-overlap ILU(0) implementation and parallel apply for AMG, but these features are still work in progress.

Starting Julia with multiple threads (for example `julia --project. --threads=4`) will allow `JutulDarcy` to make use of threads to speed up calculations

- The default behavior is to only speed up assembly of equations

- The linear solver is often the most expensive part – as mentioned above, parts can be parallelized by choosing `csr` backend when setting up the model
- Running with a parallel preconditioner can lead to higher iteration counts since the ILU(0) preconditioner changes in parallel
- Heavy compositional models benefit a lot from using threads

Threads are easy to use and can give a bit of benefit for large models.

12.1.3 Mixed-mode parallelism

You can mix the two approaches: Adding multiple threads to each MPI process can use threads to speed up assembly and property evaluations.

12.1.4 Tips for parallel runs

A few hints when you are looking at performance:

- Reservoir simulations are memory bound, cannot expect that 10 threads = 10x performance
- CPUs can often boost single-core performance when resources are available
- MPI in JutulDarcy is less tested than single-process simulations, but is natural for larger models
- There is always some cost to parallelism: If running a large ensemble with limited compute, many serial runs handled by Julia's task system is usually a better option
- Adding the maximum number of processes does not always give the best performance. Typically you want at least 10 000 cells per process. Can be case dependent.

Example: 200k cell model on laptop: 1 process 235 s -> 4 processes 145s

12.2 Solving with MPI in practice

There are a few adjustments needed before a script can be run in MPI.

12.2.1 Setting up the environment

You will have to set up an environment with the following packages under Julia 1.9+: `PartitionedArrays`, `MPI`, `JutulDarcy` and `HYPRE`. This is generally the best performing solver setup available, even if you are working in a shared memory environment.

12.2.2 Writing the script

Write your script as usual. The following options must then be set:

- setup_reservoir_model should have the extra keyword argument `split_wells=true`. We also recommend `backend=:csr` for the best performance.
- simulate_reservoir or setup_reservoir_simulator should get the optional argument `mode = :mpi`

You must then run the file using the appropriate `mpiexec` as described in the MPI.jl documentation. Specialized functions will be called by `simulate_reservoir` when this is the case. We document them here, even if we recommend using the high level version of this interface:

12.2.3 Checklist for running in MPI

- Install and load the following packages at the top of your script: `PartitionedArrays, MPI, HYPRE`
- (Recommended): Put `MPI.Init()` at the top of your script
- Make sure that `split_wells = true` is set in your model setup
- Set `output_path` when running the simulation (otherwise the results will not be stored anywhere)
- Set `mode=:mpi` when running the simulation.

A few useful functions:

- `MPI.install_mpiexecjl()` installs MPI that works "out of the box" with your current Julia setup
- `MPI.mpiexec()` gives you the path to the executable and all environment variables

A typical command to launch a MPI script from within Julia:

```
n = 5 # = 5 processes
script_to_run = "my_script.jl"
run(`$(mpiexec()) -n $n $(Base.julia_cmd()) --project=$(Base.active_project()) $script_to_run`)
```

Adding threads to the command will make JutulDarcy use both threads and processes

:warning: **Running a script in MPI means that all parts of the script will run on each process!** If you want to do data analysis you will have to either wrap your code in `if MPI.Comm_rank(MPI.COMM_WORLD) == 0` or do the data analysis in serial (recommended).

12.2.4 Limitations of running in MPI

MPI can be cumbersome to use when compared to a standard Julia script, and the current implementation relies on the model being set up on each processor before subdivision. This can be quite memory intensive during startup.

You should be familiar with the MPI programming model to use this feature. See MPI.jl for more details, and how MPI is handled in Julia specifically.

For larger models, compiling the Standalone reservoir simulator is highly recommended.

!!! note MPI consolidates results by writing files to disk. Unless you have a plan to work with the distributed states in-memory returned by the `simulate!` call, it is best to specify a `output_path` optional argument to setup_reservoir_simulator. After the simulation, that folder will contain output just as if you had run the case in serial. # GPU support

JutulDarcy includes experimental support for running linear solves on the GPU. For many simulations, the linear systems are the most compute-intensive part and a natural choice for acceleration. At the moment, the support is limited to CUDA GPUs through CUDA.jl. For the most efficient CPR preconditioner, AMGX.jl is required which is currently limited to Linux systems. Windows users may have luck by running Julia inside WSL.

12.3 How to use

If you have installed JutulDarcy, you should start by adding the CUDA and optionally the AMGX packages using the package manager:

```
using Pkg
Pkg.add("CUDA") # Requires a CUDA-capable GPU
Pkg.add("AMGX") # Requires CUDA + Linux
```

Once the packages have been added to the same environment as JutulDarcy, you can load them to enable GPU support. Let us grab the first ten steps of the EGG benchmark model:

```
using Jutul, JutulDarcy
dpth = JutulDarcy.GeoEnergyIO.test_input_file_path("EGG", "EGG.DATA")
case = setup_case_from_data_file(dpth)
case = case[1:10]
```

12.3.1 Running on CPU

If we wanted to run this on CPU we would simply call `simulate_reservoir`:

```
result_cpu = simulate_reservoir(case);
```

12.3.2 Running on GPU with block ILU(0)

If we now load `CUDA` we can run the same simulation using the CUDA-accelerated linear solver. By itself, CUDA only supports the ILU(0) preconditioner. JutulDarcy will automatically pick this preconditioner when CUDA is requested without AMGX, but we write it explicitly here:

```
using CUDA
result_ilu0_cuda = simulate_reservoir(case, linear_solver_backend = :cuda, precond = :ilu0);
```

12.3.3 Running on GPU with CPR AMGX-ILU(0)

Loading the AMGX package makes a pure GPU-based two-stage CPR available. Again, we are explicit in requesting CPR, but if both `CUDA` and `AMGX` are available and functional this is redundant:

```
using AMGX
result_amgx_cuda = simulate_reservoir(case, linear_solver_backend = :cuda, precond = :cpr);
```

In short, load `AMGX` and `CUDA` and run `simulate_reservoir(case, linear_solver_backend = :cuda)` to get GPU results. The EGG model is quite small, so if you want to see significant performance increases, a larger case will be necessary. `AMGX` also contains a large number of options that can be configured for advanced users.

12.4 Technical details and limitations

The GPU implementation relies on assembly on CPU and pinned memory to transfer onto the CPU. This means that the performance can be significantly improved by launching Julia with multiple threads to speed up the non-GPU parts of the code. AMGX is currently single-GPU only and does not work with MPI. To make use of lower precision, specify `Float32` in the `float_type` argument

to the linear solver. Additional arguments to `AMGX` can also be specified this way. For example, we can solve using aggregation AMG in single precision by doing the following:

```
simulate_reservoir(case,
    linear_solver_backend = :cuda,
    linear_solver_arg = (
        float_type = Float32,
        algorithm = "AGGREGATION",
        selector = "SIZE_8"
        )
    )
```

!!! warning “Experimental status” Multiple successive runs with different `AMGX` instances have resulted in crashes when old instances are garbage collected. This part of the code is still considered experimental, with contributions welcome if you are using it. # Standalone reservoir simulator

Scripts are interactive and useful for doing setup, simulation and post-processing in one file, but sometimes you want to run a big model unmodified from an input file:

- As an alternative to a pure Julia workflow, `JutulDarcy.jl` can be compiled into a standalone reservoir simulator
- This makes MPI simulations more ergonomic
- Compiling the code saves time when running multiple simulations
- The resulting executable is a standard command-line program - no Julia experience needed
- Output is given in the same format as regular simulations, can load data by restarting a simulation from the same `output_path`

This workflow uses PackageCompiler.jl. For more details and an example build file with keyword arguments, see the JutulDarcyApps.jl repository.

A few things to note:

- The simulator comes with a set of shared library files and will be ~500 mb
- Binaries will match platform (compiling under Linux gives you Linux binaries)
- The repository has a script that runs small “representative” models
- You can input small representative models in `precompile_jutul_darcy_mpi.jl` to make sure that compilation is avoided during simulation
- By default, the script uses the default Julia MPI binary. On a cluster, the build script may have to be modified to use the MPI type of the cluster using MPITrampoline.jl

If you get it working on a complex MPI setup, feedback on your experience and PRs are very welcome. # Package docstring

Chapter 13

Publications making use of JutulDarcy.jl

This page lists papers that use JutulDarcy.jl. Do you have a paper that is not listed here? Feel free to make a pull request, or send an e-mail with the paper title and link.

JutulDarcy.jl - a Fully Differentiable High-Performance Reservoir Simulator based on Automatic Differentiation. O. Møyner. Computational Geosciences (2025) is the main paper on JutulDarcy.jl. Please cite this if you make use of this package in your work.

13.1 Journal papers

1. Model-based reinforcement learning for active flow control. Y. Minghui, A. H. Elsheikh. Physics of Fluids 37.9 (2025)
2. Grid-Orientation Effects in the 11th SPE Comparative Solution Project Using Unstructured Grids and Consistent Discretizations. K. Holme, K.-A. Lie, O. Møyner, A. Johansson. SPE Journal (2025)
3. Diffusion-Based Subsurface Multiphysics Monitoring and Forecasting. X. Huang, F. Wang, T. Alkhalifah. JGR Machine Learning And Computation (2025)
4. Accelerating Nonlinear Convergence in Reservoir Simulation by Adaptive Relaxation and Nonlinear Domain-Decomposition Preconditioning. K-A. Lie, O. Møyner, Ø.Klemetsdal. SPE Journal (2025)
5. An uncertainty-aware digital shadow for underground multimodal CO2 storage monitoring. A.P. Gahlot, R. Orozco, Z. Yin, G. Bruer, F.J. Herrmann. Geophysical Journal International (2025)
6. Nonlinear domain-decomposition preconditioning for robust and efficient field-scale simulation of subsurface flow. O. Møyner, Atgeirr F. Rasmussen, Ø. Klemetsdal, H.M. Nilsen, A. Moncorgé, and K-A. Lie. Computational Geosciences (2024)
7. Time-lapse full-waveform permeability inversion: A feasibility study. Z. Yin, M. Louboutin, O. Møyner, F.J. Herrmann. The Leading Edge (2024)
8. Seismic Monitoring of CO Plume Dynamics Using Ensemble Kalman Filtering. G. Bruer, A.P. Gahlot, E. Chow, Felix Herrmann. IEEE Transactions on Geoscience and Remote Sensing (2024)

9. Learned multiphysics inversion with differentiable programming and machine learning. M. Louboutin, Z. Yin, R. Orozco, Thomas J. Grady II, Ali Siahkoohi, Gabrio Rizzuti, Philipp A. Witte, O. Møyner, G.J. Gorman, and F.J. Herrmann, The Leading Edge 2023 42:7, 474-486 (2023)
10. Solving multiphysics-based inverse problems with learned surrogates and constraints. Z. Yin, R. Orozco, M. Louboutin & F.J. Herrmann. Advanced Modeling and Simulation in Engineering Sciences (2023)

13.2 In proceedings

1. Enabling Intelligent Reservoir Engineer: Exploring New Possibilities with Reinforcement Learning and Proxy Models. X. Wang, J. Li, J. Shu, Y. Cui. SPE Advances in Integrated Reservoir Modelling and Field Development Conference and Exhibition (2025)
2. Reduced physics-based simulation for unconventional production forecasting – A 1D approach. S. Krogstad, M.A. Jakymec, A. Kianinejad, D. Pertuso, S. Matringe, A. Brostrom, J. Torben, O. Møyner, and K.-A. Lie. URTEC (2025)
3. Predictive Digital Twins for Underground Thermal Energy Storage using Differentiable Programming. Ø. Klemetsdal, O. Andersen, S. Krogstad. DTE - AICOMAS (2025)
4. Sensitivity-aware rock physics enhanced digital shadow for underground-energy storage monitoring. A.P. Gahlot, H.T. Erdinc, F.J. Herrmann. International Meeting for Applied Geoscience and Energy (2025)
5. A reduced-order derivative-informed neural operator for subsurface fluid-flow. International Meeting for Applied Geoscience and Energy. J. Park, G. Bruer, H. Erdinc, A. Gahlot, F.J. Herrmann (2025)
6. A Data-Driven Approach to Select Optimal Time Steps for Complex Reservoir Models. O. Møyner, K-A. Lie. SPE Reservoir Simulation Conference (2025)
7. An Accurate and Efficient Surrogate Model-Based Framework for Coupled Wellbore-Reservoir Modeling. Jin S., Guoqing H., Zhenduo Y., Xin W., Long P., Junjian L. Advances in Integrated Reservoir Modelling and Field Development Conference and Exhibition (2025)
8. JutulDarcy.jl - a Fully Differentiable High-Performance Reservoir Simulator based on Automatic Differentiation. O. Møyner. ECMOR 2024 (2024)
9. Proxy Models for Rapid Simulation of Underground Thermal Energy Storage. Ø. Klemetsdal, O. Andersen. EAGE GET (2024)
10. Enhancing Performance of Complex Reservoir Models via Convergence Monitors. K-A. Lie, O. Møyner, Ø. Klemetsdal, B. Skaflestad, A. Moncorgé and V. Kippe, ECMOR 2024 (2024)
11. Inference of CO2 flow patterns – a feasibility study. A.P. Gahlot, H.T- Erdinc, R- Orozco, Z. Yin, F.J. Herrmann. NeurIPS 2023 Workshop - Tackling Climate Change with Machine Learning (2023)
12. Well Control Optimization with Output Constraint Handling by Means of a Derivative-Free Trust Region Algorithm. M. Hannanu, T. L. Silva, E. Camponogara, M. Hovd. ADIPEC (2023)
13. An Adaptive Newton–ASPEN Solver for Complex Reservoir Models. K-A. Lie, O. Møyner and Ø. Klemetsdal. SPE Reservoir Simulation Conference, Galveston, Texas, USA, March (2023)

13.3 Preprints

1. Benchmarking CO Storage Simulations: Results from the 11th Society of Petroleum Engineers Comparative Solution Project. J.M. Nordbotten, M.A. Fernø, B. Flemisch, A.R. Kovscek, K.-A. Lie, J.W. Both, O. Møyner, T.H. Sandve, E. Ahusborde, S. Bauer, Z. Chen, H. Class, C. Di, D. Ding, D. Element, A. Firoozabadi, E. Flauraud, J. Franc, F. Gasanzade, Y. Ghomian, M.A. Giddins, C. Green, B.R.B. Fernandes, G. Hadjisotiriou, G. Hammond, H. Huang, D. Kachuma, M. Kern, T. Koch, P. Krishnamurthy, K.O. Lye, D. Landa-Marbán, M. Nole, P. Orsini, N. Ruby, P. Salinas, M. Sayyafzadeh, J. Solovský, J. Torben, A. Turner, D.V. Voskov, K. Wendel, A.A. Youssef (2025)
2. Advancing geological carbon storage monitoring with 3D digital shadow technology. A.P. Gahlot, R. Orozco, F.J. Herrmann (2025)
3. Enhancing robustness of digital shadow for CO2 storage monitoring with augmented rock physics modeling. A.P. Gahlot, F.J. Herrmann (2025)
4. A digital twin for geological carbon storage with controlled injectivity. A.P. Gahlot, H. Li, Z. Yin, R. Orozco, F.J. Herrmann (2024)
5. Well2Flow: Reconstruction of reservoir states from sparse wells using score-based generative models. S. Zeng, H. Li, A.P. Gahlot, F.J. Herrmann. International Meeting for Applied Geoscience and Energy (2025)

13.4 Theses

1. Grid Orientation Effects and Consistent Discretizations for Simulation of Geologic Carbon Storage: A Study of the SPE11 Benchmark. K. Holme. MSc thesis, NTNU (2024) # Documentation from Jutul.jl

`JutulDarcy.jl` builds upon `Jutul.jl`, which takes care of the heavy lifting in terms of meshes, discretizations and solvers. You can use `JutulDarcy.jl` without knowing the inner workings of `Jutul.jl`, but if you want to dive under the hood the Jutul.jl manual and Jutul.jl docstrings may be useful.

We include the docstrings here for your convenience:

Chapter 14

Example Overview

JutulDarcy.jl comes with a number of examples that illustrate different features of the simulator.

The examples are organized by category. For the full interactive examples with code and outputs, please visit the online documentation at https://sintefmath.github.io/JutulDarcy.jl/

14.1 Introduction

Basic examples that illustrate fundamental features of JutulDarcy.jl.

- **Afi Input File** (`afi_input_file.jl`)
- **Data Input File** (`data_input_file.jl`)
- **Intro Sensitivities** (`intro_sensitivities.jl`)
- **Spe10** (`spe10.jl`)
- **Two Phase Buckley Leverett** (`two_phase_buckley_leverett.jl`)
- **Two Phase Gravity Segregation** (`two_phase_gravity_segregation.jl`)
- **Two Phase Unstable Gravity** (`two_phase_unstable_gravity.jl`)
- **Wells Intro** (`wells_intro.jl`)

14.2 Workflow

Examples demonstrating complete workflows and advanced use cases.

- **Adding New Wells** (`adding_new_wells.jl`)
- **Co2 Sloped** (`co2_sloped.jl`)
- **Equilibrium State** (`equilibrium_state.jl`)
- **Five Spot Ensemble** (`five_spot_ensemble.jl`)
- **Fully Differentiable Geothermal** (`fully_differentiable_geothermal.jl`)
- **Hybrid Simulation Relperm** (`hybrid_simulation_relperm.jl`)
- **Model Coarsening** (`model_coarsening.jl`)
- **Rate Optimization** (`rate_optimization.jl`)
- **Tracers Two Wells** (`tracers_two_wells.jl`)

14.3 Data Assimilation

Examples of history matching, optimization, and sensitivity analysis.

- **Advanced History Match** (`advanced_history_match.jl`)
- **Cgnet Egg** (`cgnet_egg.jl`)
- **Optimize Simple Bl** (`optimize_simple_bl.jl`)
- **Spe1 Gradients** (`spe1_gradients.jl`)

14.4 Geothermal

Geothermal reservoir simulation examples.

- **Geothermal Doublet** (`geothermal_doublet.jl`)
- **Htates Intro** (`htates_intro.jl`)

14.5 Compositional

Compositional flow and multi-component examples.

- **Clapeyron** (`clapeyron.jl`)
- **Compositional 2D Vertical** (`compositional_2d_vertical.jl`)
- **Compositional 5Components** (`compositional_5components.jl`)

14.6 Discretization

Examples showing different discretization schemes.

- **Consistent Avgmpfa** (`consistent_avgmpfa.jl`)
- **Mpfa Weno Discretizations** (`mpfa_weno_discretizations.jl`)

14.7 Properties

Examples focusing on fluid properties and relationships.

- **Co2 Props** (`co2_props.jl`)
- **Relperms** (`relperms.jl`)

14.8 Validation

Validation cases comparing with other simulators and benchmarks.

- **Validation Compositional** (`validation_compositional.jl`)
- **Validation Egg** (`validation_egg.jl`)
- **Validation Mrst** (`validation_mrst.jl`)
- **Validation Norne Nohyst** (`validation_norne_nohyst.jl`)
- **Validation Olympus 1** (`validation_olympus_1.jl`)
- **Validation Polymer** (`validation_polymer.jl`)
- **Validation Spe1** (`validation_spe1.jl`)

- **Validation Spe9** (`validation_spe9.jl`)
- **Validation Thermal** (`validation_thermal.jl`)

Note: The full examples with code, plots, and detailed explanations are available in the online documentation. You can also find the example scripts in the `examples/` directory of the repository. # Set up a .AFI file for simulation

JutulDarcy has partial support for setting up and reading the .AFI file format. This example demonstrates how to load AFI files. ## Load the OLYMPUS model We first load the Olympus model from a RESQML-based AFI file.

```
using Jutul, JutulDarcy, GeoEnergyIO
fn = GeoEnergyIO.test_input_file_path("OLYMPUS_25_AFI_RESQML", "OLYMPUS_25.afi")
case_olympus = setup_case_from_afi(fn);
```

14.9 Reading files with a pre-defined reservoir

Note that as the AFI support requires either inline mesh definitions or RESQML, and as such, not all files can be read directly. However, if we have already set up a reservoir model (for example from a DATA file), we can reuse the mesh and mesh properties from that model when setting up the AFI case. We set up SPE9 with a pre-defined reservoir model, bypassing the need for GSG support.

```
spe9_dir = JutulDarcy.GeoEnergyIO.test_input_file_path("SPE9")
case = setup_case_from_data_file(joinpath(spe9_dir, "SPE9.DATA"))
reservoir = reservoir_domain(case)
fn = GeoEnergyIO.test_input_file_path("SPE9_AFI_GSG", "SPE9_clean_split.afi")
case_ix = setup_case_from_afi(fn, reservoir = reservoir);
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Simulating Eclipse/DATA input files The DATA format is commonly used in reservoir simulation. JutulDarcy can set up cases on this format and includes a fully featured grid builder for corner-point grids. Once a case has been set up, it uses the same types as a regular JutulDarcy simulation, allowing modification and use of the case in differentiable workflows.

We begin by loading the SPE9 dataset via the GeoEnergyIO package. This package includes a set of open datasets that can be used for testing and benchmarking. The SPE9 dataset is a 3D model with a corner-point grid and a set of wells produced by the Society of Petroleum Engineers. The specific version of the file included here is taken from the OPM tests repository.

```
using JutulDarcy, GeoEnergyIO
pth = GeoEnergyIO.test_input_file_path("SPE9", "SPE9.DATA");
```

14.10 Set up and run a simulation

We have supressed the output of the simulation to avoid cluttering the documentation, but we can set the `info_level` to a higher value to see the output.

If we do not need the case, we could also have simulated by passing the path: `ws, states = simulate_data_file(pth)`

```
case = setup_case_from_data_file(pth)
ws, states = simulate_reservoir(case);
```

14.11 Show the input data

The input data takes the form of a Dict:

```
case.input_data
```

We can also examine the for example RUNSPEC section, which is also represented as a Dict.

```
case.input_data["RUNSPEC"]
```

14.12 Plot the simulation model

These plot are normally interactive, but if you are reading the published online documentation static screenshots will be inserted instead.

```
using GLMakie
plot_reservoir(case.model, states)
```

14.13 Plot the well responses

We can plot the well responses (rates and pressures) in an interactive viewer. Multiple wells can be plotted simultaneously, with options to select which units are to be used for plotting.

```
plot_well_results(ws)
```

14.14 Plot the field responses

Similar to the wells, we can also plot field-wide measurables. We plot the field gas production rate and the average pressure as the initial selection. If you are running this case interactively you can select which measurables to plot.

We observe that the field pressure steadily decreases over time, as a result of the gas production. The drop in pressure is not uniform, as during the period where little gas is produced, the decrease in field pressure is slower.

```
plot_reservoir_measurables(case, ws, states, left = :fgpr, right = :pres)
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Intro to sensitivities in JutulDarcy Sensitivites with respect to custom parameters: We demonstrate how to set up a simple conceptual model, add new parameters and variable definitions in the form of a new relative permeability function, and calculate and visualize parameter sensitivities.

We first set up a quarter-five-spot model where the domain is flooded from left to right. Some cells have lower permeability to impede flow and make the scenario more interesting.

For more details, see the paper JutulDarcy.jl - a Fully Differentiable High-Performance Reservoir Simulator Based on Automatic Differentiation.

```
using Jutul, JutulDarcy, GLMakie, HYPRE
darcy, kg, meter, year, day, bar = si_units(:darcy, :kilogram, :meter, :year, :day, :bar)

L = 1000.0meter
H = 100.0meter
big = false # Paper uses big, takes some more time to run
if big
    nx = 500
else
    nx = 100
end
dx = L/nx

g = CartesianMesh((nx, nx, 1), (L, L, H))
nc = number_of_cells(g)
perm = fill(0.1darcy, nc)

reservoir = reservoir_domain(g, permeability = 0.1darcy)
centroids = reservoir[:cell_centroids]
rock_type = fill(1, nc)
for (i, x, y) in zip(eachindex(perm), centroids[1, :], centroids[2, :])
    xseg = (x > 0.2L) & (x < 0.8L) & (y > 0.75L) & (y < 0.8L)
    yseg = (y > 0.2L) & (y < 0.8L) & (x > 0.75L) & (x < 0.8L)
    if xseg || yseg
        rock_type[i] = 2
    end
    xseg = (x > 0.2L) & (x < 0.55L) & (y > 0.50L) & (y < 0.55L)
    yseg = (y > 0.2L) & (y < 0.55L) & (x > 0.50L) & (x < 0.55L)
    if xseg || yseg
        rock_type[i] = 3
    end
    xseg = (x > 0.2L) & (x < 0.3L) & (y > 0.25L) & (y < 0.3L)
    yseg = (y > 0.2L) & (y < 0.3L) & (x > 0.25L) & (x < 0.3L)
    if xseg || yseg
        rock_type[i] = 4
    end
end
```

```
perm = reservoir[:permeability]
@. perm[rock_type == 2] = 0.001darcy
@. perm[rock_type == 3] = 0.005darcy
@. perm[rock_type == 4] = 0.01darcy

I = setup_vertical_well(reservoir, 1, 1, name = :Injector)
P = setup_vertical_well(reservoir, nx, nx, name = :Producer)

phases = (AqueousPhase(), VaporPhase())
rhoWS, rhoGS = 1000.0kg/meter^3, 700.0kg/meter^3
system = ImmiscibleSystem(phases, reference_densities = (rhoWS, rhoGS))

model = setup_reservoir_model(reservoir, system, wells = [I, P])
rmodel = reservoir_model(model)
```

14.15 Plot the initial variable graph

We plot the default variable graph that describes how the different variables relate to each other. When we add a new parameter and property in the next section, the graph is automatically modified.

```
using NetworkLayout, LayeredLayouts, GraphMakie
Jutul.plot_variable_graph(rmodel)
```

14.16 Change the variables

We replace the density variable with a more compressible version, and we also define a new relative permeability variable that depends on a new parameter `KrExponents` to define the exponent of the relative permeability in each cell and phase of the model.

This is done through several steps: 1. First, we define the type 2. We define functions that act on that type, in particular the update function that is used to evaluate the new relative permeability during the simulation for named inputs `Saturations` and `KrExponents`. 3. We define the `KrExponents` as a model parameter with a default value, that can subsequently be used by the relative permeability.

Finally we plot the variable graph again to verify that the new relationship has been included in our model.

```
c = [1e-6/bar, 1e-4/bar]
density = ConstantCompressibilityDensities(p_ref = 1*bar, density_ref = [rhoWS, rhoGS], compres
replace_variables!(rmodel, PhaseMassDensities = density);

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

```
            n_  = KrExponents[ph, c]
            vals[ph, c] = S_ ^n_
        end
    end
end
struct MyKrExp <: PhaseVariables end
Jutul.default_value(model, ::MyKrExp) = 2.0
set_parameters!(rmodel, KrExponents = MyKrExp())
replace_variables!(rmodel, RelativePermeabilities = MyKr());
Jutul.plot_variable_graph(rmodel)
```

14.17 Set up scenario and simulate

```
parameters = setup_parameters(model)
exponents = parameters[:Reservoir][:KrExponents]
for (cell, rtype) in enumerate(rock_type)
    if rtype == 1
        exp_w = 2
        exp_g = 3
    else
        exp_w = 1
        exp_g = 2
    end
    exponents[1, cell] = exp_w
    exponents[2, cell] = exp_g
end

pv = pore_volume(model, parameters)
state0 = setup_reservoir_state(model, Pressure = 150*bar, Saturations = [1.0, 0.0])

dt = repeat([30.0]*day, 12*5)
pv = pore_volume(model, parameters)
total_time = sum(dt)
inj_rate = sum(pv)/total_time

rate_target = TotalRateTarget(inj_rate)
I_ctrl = InjectorControl(rate_target, [0.0, 1.0], density = rhoGS)
bhp_target = BottomHolePressureTarget(50*bar)
P_ctrl = ProducerControl(bhp_target)
controls = Dict()
controls[:Injector] = I_ctrl
controls[:Producer] = P_ctrl

forces = setup_reservoir_forces(model, control = controls)
case = JutulCase(model, dt, forces, parameters = parameters, state0 = state0)
result = simulate_reservoir(case, output_substates = true);
```

14.18 Print the gas saturation

```
ws, states = result
ws(:Producer, :grat)
```

14.19 Define objective function

We let the objective function be the amount produced of produced gas, normalized by the injected amount.

```
using GLMakie
function objective_function(model, state, Δt, step_i, forces)
    grat = JutulDarcy.compute_well_qoi(model, state, forces, :Producer, SurfaceGasRateTarget)
    return Δt*grat/(inj_rate*total_time)
end
data_domain_with_gradients = JutulDarcy.reservoir_sensitivities(case, result, objective_functi
```

14.20 Launch interactive plotter for cell-wise gradients

```
plot_reservoir(data_domain_with_gradients)
```

14.21 Set up plotting functions

```
 K = data_domain_with_gradients[:permeability]
   = data_domain_with_gradients[:porosity]

function get_cscale(x)
    minv0, maxv0 = extrema(x)
    minv = min(minv0, -maxv0)
    maxv = max(maxv0, -minv0)
    return (minv, maxv)
end

function myplot(title, vals; kwarg...)
    fig = Figure()
    myplot!(fig, 1, 1, title, vals; kwarg...)
    return fig
end

function myplot!(fig, I, J, title, vals; is_grad = false, is_log = false, colorrange = missing
    ax = Axis(fig[I, J], title = title)

    if is_grad
        if ismissing(colorrange)
            colorrange = get_cscale(vals)
```

```
        end
        cmap = :seismic
    else
        if ismissing(colorrange)
            colorrange = extrema(vals)
        end
        cmap = :seaborn_icefire_gradient
    end
    hidedecorations!(ax)
    hidespines!(ax)
    arg = (; colormap = cmap, colorrange = colorrange, kwarg...)
    plt = plot_cell_data!(ax, g, vals; shading = NoShading, arg...)
    if colorbar
        if ismissing(ticks)
            ticks = range(colorrange..., nticks)
        end
        Colorbar(fig[I, J+1], plt, ticks = ticks, ticklabelsize = 25, size = 25)
    end
    return fig
end
```

14.22 Plot the permeability

```
myplot("Permeability", perm./darcy, colorscale = log10, ticks = [0.001, 0.01, 0.1])
```

14.23 Plot the evolution of the gas saturation

```
fig = Figure(size = (1200, 400))
sg = states[25][:Saturations][2, :]
myplot!(fig, 1, 1, "Gas saturation", sg, colorrange = (0, 1), colorbar = false)
sg = states[70][:Saturations][2, :]
myplot!(fig, 1, 2, "Gas saturation", sg, colorrange = (0, 1), colorbar = false)
sg = states[end][:Saturations][2, :]
myplot!(fig, 1, 3, "Gas saturation", sg, colorrange = (0, 1))
fig
```

14.24 Plot the sensitivity of the objective with respect to permeability

```
if big
    cr = (-0.001, 0.001)
    cticks = [-0.001, -0.0005, 0.0005, 0.001]
else
    cr = (-0.05, 0.05)
```

```
    cticks = [-0.05, -0.025, 0, 0.025, 0.05]
end

myplot("perm_sens",  K.*darcy, is_grad = true, ticks = cticks, colorrange = cr)
```

14.25 Plot the sensitivity of the objective with respect to porosity

```
if big
    cr = (-0.00001, 0.00001)
else
    cr = (-0.00025, 0.00025)
end
myplot("porosity_sens",  , is_grad = true, colorrange = cr)
```

14.26 Gradient with respect to cell centroids

```
xyz = data_domain_with_gradients[:cell_centroids]
x =  xyz[1, :]
y =  xyz[2, :]
z =  xyz[3, :]
##
if big
    cr = [-1e-8, 1e-8]
else
    cr = [-1e-7, 1e-7]
end
```

14.27 Plot the sensitivity of the objective with respect to x cell centroids

```
myplot("dx_sens",  x, is_grad = true, colorrange = cr)
```

14.28 Plot the sensitivity of the objective with respect to y cell centroids

```
myplot("dy_sens",  y, is_grad = true, colorrange = cr)
```

14.29 Plot the sensitivity of the objective with respect to z cell centroids

Note: The effect here is primarily coming from gravity.

```
myplot("dz_sens",  z, is_grad = true, colorrange = cr)
```

14.30 Plot the effect of the new liquid kr exponent on the gas production

```
if big
    cr = [-1e-7, 1e-7]
else
    cr = [-8e-6, 8e-6]
end

kre = data_domain_with_gradients[:KrExponents]
exp_l = kre[1, :]
myplot("exp_liquid", exp_l, is_grad = true, colorrange = cr)
```

14.31 Plot the effect of the new vapor kr exponent on the gas production

```
exp_v = kre[2, :]
myplot("exp_vapor", exp_v, is_grad = true, colorrange = cr)
```

14.32 Plot the effect of the liquid phase viscosity

Note: The viscosity can in many models be a variable and not a parameter. For this simple model, however, it is treated as a parameter and we obtain sensitivities.

```
mu = data_domain_with_gradients[:PhaseViscosities]
if big
    cr = [-0.001, 0.001]
else
    cr = [-0.01, 0.01]
end
mu_l = mu[1, :]
myplot("mu_liquid", mu_l, is_grad = true, colorrange = cr)
```

14.33 Plot the effect of the liquid phase viscosity

```
mu_v = mu[2, :]
myplot("mu_vapor", mu_v, is_grad = true, colorrange = cr)
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # SPE10, model

2 The SPE10 benchmark case is a standard benchmark case for reservoir simulators. The model is described in detail in the SPE10 benchmark page

This example demonstrates routines to set up the SPE10, model 2, case using premade functions in the `JutulDarcy.SPE10` module. This model is often just called SPE10, since the first model of the benchmark is very small. ## Set up the reservoir We can set up the reservoir itself to have a look at the static properties.

```
using JutulDarcy, GLMakie, HYPRE
reservoir = JutulDarcy.SPE10.setup_reservoir()
plot_reservoir(reservoir, key = :porosity)
```

14.34 Set up and run a simulation for the first layer

We can set up and run a full simulation for the first layer of the model. As we want the example to run quickly, we just pick the top layer. The function is set up to scale the default well rates to the smaller model, so you can adjust the number of layers without changing anything else.

```
case = JutulDarcy.SPE10.setup_case(layers = 1:1)
```

14.34.1 Plot the porosity

Note that some cells are removed due to very low porosity. The setup function has additional options to control this behavior if you would rather limit the minimum porosity than removing cells.

```
plot_reservoir(case.model, key = :porosity)
```

14.34.2 Run the simulation

```
ws, states = simulate_reservoir(case)
```

14.34.3 Plot the final saturation

```
plot_reservoir(case.model, states, key = :Saturations, step = length(states))
```

14.35 Show the last layer

The first 35 layers correspond to the Tarbert formation and the latter 50 layers are the upper ness formation. We plot one of the last layers, showing a channelized, fluvial structure.

```
reservoir_ness = JutulDarcy.SPE10.setup_reservoir(layers = 60)
plot_reservoir(reservoir_ness, key = :porosity)
```

14.36 Coarsen the model

The full model is quite large, with 1.1 million cells. We can coarsen the model to get a smaller model that is faster to simulate. The original model was intended as a benchmark for upscaling

methods, even though such models are now quite easy to simulate when using parallel computing on CPU/GPU.

The default coarsening is quite simple (harmonic averages and sums), but quickly sets up a coarse model that can be used for testing, or as a starting point for a more refined coarsening.

```
case_fine = JutulDarcy.SPE10.setup_case()
case_coarse = coarsen_reservoir_case(case_fine, (10, 22, 40))
```

14.36.1 Simulate the coarse model

The now quite coarse model should run a few seconds.

```
ws_coarse, states_coarse = simulate_reservoir(case_coarse);
plot_reservoir(case_coarse.model, states_coarse, key = :Saturations, step = length(states_coars
```

14.36.2 Plot the water cut

```
using Jutul
t = ws_coarse.time./si_unit(:day)
fig = Figure()
ax = Axis(fig[1, 1]; xlabel = "Time (days)", ylabel = "Water cut")
for w in [:P1, :P2, :P3, :P4]
    wc = ws_coarse[w, :wcut]
    lines!(ax, t, wc; label = "$w")
end
axislegend(position = :lt)
fig
```

14.37 Conclusion

The SPE10, model 2, case is a standard benchmark case for reservoir simulators. JutulDarcy includes premade functions to set up and run this model, including wells, PVT, and the schedule from the original case. The model can be run as-is, or coarsened to a smaller size for testing and development.

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Buckley-Leverett two-phase problem The Buckley-Leverett test problem is a classical reservoir simulation benchmark that demonstrates the nonlinear displacement process of a viscous fluid being displaced by a less viscous fluid, typically taken to be water displacing oil.

14.38 Problem definition

This is a simple model without wells, where the flow is driven by a simple source term and a simple constant pressure boundary condition at the outlet. We define a function that sets up a two-phase

system, a simple 1D domain and replaces the default relative permeability functions with quadratic functions:

```
k_{r\alpha}(S) = \min \left(\frac{S - S_r}{1 - S_r}, 1\right)^n, S_r = 0.2, n = 2
```

In addition, the phase viscosities are treated as constant parameters of 1 and 5 centipoise for the displacing and resident fluids, respectively.

The function is parametrized on the number of cells and the number of time-steps used to solve the model. This function, since it uses a relatively simple setup without wells, uses the `Jutul` functions directly.

```
using JutulDarcy, Jutul
function solve_bl(;nc = 100, time = 1.0, nstep = nc)
    T = time
    tstep = repeat([T/nstep], nstep)
    domain = get_1d_reservoir(nc)
    nc = number_of_cells(domain)
    timesteps = tstep*3600*24
    bar = 1e5
    p0 = 100*bar
    sys = ImmiscibleSystem((LiquidPhase(), VaporPhase()))
    model = SimulationModel(domain, sys)
    kr = BrooksCoreyRelativePermeabilities(sys, [2.0, 2.0], [0.2, 0.2])
    replace_variables!(model, RelativePermeabilities = kr)
    tot_time = sum(timesteps)
    pv = pore_volume(domain)
    irate = 500*sum(pv)/tot_time
    src  = SourceTerm(1, irate, fractional_flow = [1.0, 0.0])
    bc = FlowBoundaryCondition(nc, p0/2)
    forces = setup_forces(model, sources = src, bc = bc)
    parameters = setup_parameters(model, PhaseViscosities = [1e-3, 5e-3]) # 1 and 5 cP
    state0 = setup_state(model, Pressure = p0, Saturations = [0.0, 1.0])
    states, report = simulate(state0, model, timesteps,
        forces = forces, parameters = parameters)
    return states, model, report
end
```

14.39 Run the base case

We solve a small model with 100 cells and 100 steps to serve as the baseline.

```
n, n_f = 100, 1000
states, model, report = solve_bl(nc = n)
print_stats(report)
```

14.40 Run refined version (1000 cells, 1000 steps)

Using a grid with 100 cells will not yield a fully converged solution. We can increase the number of cells at the cost of increasing the runtime a bit. Note that most of the time is spent in the linear solver, which uses a direct sparse LU factorization by default. For larger problems it is recommended to use an iterative solver. The high-level interface used in later examples automatically sets up an iterative solver with the appropriate preconditioner.

```
states_refined, _, report_refined = solve_bl(nc = n_f);
print_stats(report_refined)
```

14.41 Plot results

We plot the saturation front for the base case at different times together with the final solution for the refined model. In this case, refining the grid by a factor 10 gave us significantly less smearing of the trailing front.

```
using GLMakie
x = range(0, stop = 1, length = n)
x_f = range(0, stop = 1, length = n_f)
f = Figure()
ax = Axis(f[1, 1], ylabel = "Saturation", title = "Buckley-Leverett displacement")
for i in 1:6:length(states)
    lines!(ax, x, states[i][:Saturations][1, :], color = :darkgray)
end
lines!(ax, x_f, states_refined[end][:Saturations][1, :], color = :red)
f
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Gravity segregation example The simplest type of porous media simulation problem to set up that is not trivial is the transition to equilibrium from an unstable initial condition. Placing a heavy fluid on top of a lighter fluid will lead to the heavy fluid moving down while the lighter fluid moves up.

14.42 Problem set up

We define a simple 1D gravity column with an approximate 10-1 ratio in density between the two compressible phases and let it simulate until equilibrium is reached. We begin by defining the reservoir domain itself.

```
using JutulDarcy, Jutul
nc = 100
Darcy, bar, kg, meter, day = si_units(:darcy, :bar, :kilogram, :meter, :day)

g = CartesianMesh((1, 1, nc), (1.0, 1.0, 10.0))
domain = reservoir_domain(g, permeability = 1.0*Darcy);
```

14.43 Fluid properties

Define two phases liquid and vapor with a 10-1 ratio reference densities and set up the simulation model.

```
p0 = 100*bar

rhoLS = 1000.0*kg/meter^3
rhoVS = 100.0*kg/meter^3
cl, cv = 1e-5/bar, 1e-4/bar
L, V = LiquidPhase(), VaporPhase()
sys = ImmiscibleSystem([L, V])
model = SimulationModel(domain, sys);
```

14.43.1 Definition for phase mass densities

Replace default density with a constant compressibility function that uses the reference values at the initial pressure.

```
density = ConstantCompressibilityDensities(sys, p0, [rhoLS, rhoVS], [cl, cv])
set_secondary_variables!(model, PhaseMassDensities = density);
```

14.43.2 Set up initial state

Put heavy phase on top and light phase on bottom. Saturations have one value per phase, per cell and consequently a per-cell instantiation will require a two by number of cells matrix as input. We also set up time-steps for the simulation, using the provided conversion factor to convert days into seconds.

```
nl = nc ÷ 2
sL = vcat(ones(nl), zeros(nc - nl))'
s0 = vcat(sL, 1 .- sL)
state0 = setup_state(model, Pressure = p0, Saturations = s0)
timesteps = repeat([0.02]*day, 150);
```

14.44 Perform simulation

We simulate the system using the default linear solver and otherwise default options. Using `simulate` with the default options means that no dynamic timestepping will be used, and the simulation will report on the exact 150 steps defined above.

```
states, report = simulate(state0, model, timesteps);
```

14.45 Plot results

Plot the saturations of the liquid phase at three different timesteps: The initial, unstable state, an intermediate state where fluid exchange between the top and bottom is initiated, and the final equilibrium state where the phases have swapped places.

```
using GLMakie
fig = Figure()
function plot_sat!(ax, state)
    plot_cell_data!(ax, g, state[:Saturations][1, :],
        colorrange = (0.0, 1.0),
        colormap = :seismic
    )
end
ax1 = Axis3(fig[1, 1], title = "Initial state", aspect = (1, 1, 4.0))
plot_sat!(ax1, state0)

ax2 = Axis3(fig[1, 2], title = "Intermediate state", aspect = (1, 1, 4.0))
plot_sat!(ax2, states[25])

ax3 = Axis3(fig[1, 3], title = "Final state", aspect = (1, 1, 4.0))
plt = plot_sat!(ax3, states[end])
Colorbar(fig[1, 4], plt)
fig
```

14.45.1 Plot time series

The 1D nature of the problem allows us to plot all timesteps simultaneously in 2D. We see that the heavy fluid, colored blue, is initially at the top of the domain and the lighter fluid is at the bottom. These gradually switch places until all the heavy fluid is at the lower part of the column.

```
tmp = vcat(map((x) -> x[:Saturations][1, :]', states)...)
f = Figure()
ax = Axis(f[1, 1], xlabel = "Time", ylabel = "Depth", title = "Gravity segregation")
hm = heatmap!(ax, tmp, colormap = :seismic)
Colorbar(f[1, 2], hm)
f
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Gravity circulation with CPR preconditioner This example demonstrates a more complex gravity driven instability. The problem is a bit larger than the Gravity segregation example, and is therefore set up using the high level API that automatically sets up an iterative linear solver with a constrained pressure residual (CPR) preconditioner and automatic timestepping.

The high level API uses the more low level `Jutul` API seen in the other examples under the hood and makes more complex problems easy to set up. The same data structures and functions are used, allowing for deep customization if the defaults are not appropriate.

```
using JutulDarcy
using Jutul
using GLMakie
cmap = :seismic
nx = nz = 100;
```

14.46 Define the domain

```
D = 10.0
g = CartesianMesh((nx, 1, nz), (D, 1.0, D))
domain = reservoir_domain(g);
```

14.47 Set up model and properties

```
Darcy, bar, kg, meter, day = si_units(:darcy, :bar, :kilogram, :meter, :day)
p0 = 100*bar
rhoLS = 1000.0*kg/meter^3 # Definition of fluid phases
rhoVS = 500.0*kg/meter^3
cl, cv = 1e-5/bar, 1e-4/bar
L, V = LiquidPhase(), VaporPhase()
sys = ImmiscibleSystem([L, V])
model = setup_reservoir_model(domain, sys)
parameters = setup_parameters(model)
density = ConstantCompressibilityDensities(sys, p0, [rhoLS, rhoVS], [cl, cv]) # Replace densit
replace_variables!(model, PhaseMassDensities = density);
kr = BrooksCoreyRelativePermeabilities(sys, [2.0, 3.0])
replace_variables!(model, RelativePermeabilities = kr)
```

14.47.1 Define initial saturation

Set the left part of the domain to be filled by the vapor phase and the heavy liquid phase in the remainder. To do this, we grab the cell centroids in the x direction from the domain, reshape them to the structured mesh we are working on and define the liquid saturation from there.

```
c = domain[:cell_centroids]
x = reshape(c[1, :], nx, nz)

sL = zeros(nx, nz)
plane = D/2.0
for i in 1:nx
    for j = 1:nz
        X = x[i, j]
        sL[i, j] = clamp(Float64(X > plane), 0, 1)
    end
end
heatmap(sL, colormap = cmap, axis = (title = "Initial saturation",))
```

14.47.2 Set up initial state

```
sL = vec(sL)'
sV = 1 .- sL
s0 = vcat(sV, sL)
```

```
state0 = setup_reservoir_state(model, Pressure = p0, Saturations = s0)
```

14.47.3 Set the viscosity of the phases

By default, viscosity is a parameter and can be set per-phase and per cell.

```
  = parameters[:Reservoir][:PhaseViscosities]
@.  [1, :] = 1e-3
@.  [2, :] = 5e-3
```

Convert time-steps from days to seconds

```
timesteps = repeat([10.0*3600*24], 20)
_, states, = simulate_reservoir(state0, model, timesteps, parameters = parameters);
```

14.48 Plot results

Plot initial saturation

```
tmp = reshape(state0[:Reservoir][:Saturations][1, :], nx, nz)
f = Figure()
ax = Axis(f[1, 1], title = "Before")
heatmap!(ax, tmp, colormap = cmap)
```

Plot intermediate saturation

```
tmp = reshape(states[length(states) ÷ 2][:Saturations][1, :], nx, nz)
ax = Axis(f[1, 2], title = "Half way")
hm = heatmap!(ax, tmp, colormap = cmap)
```

Plot final saturation

```
tmp = reshape(states[end][:Saturations][1, :], nx, nz)
ax = Axis(f[1, 3], title = "After")
hm = heatmap!(ax, tmp, colormap = cmap)
Colorbar(f[1, 4], hm)
f
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Introduction to wells This example demonstrates how to set up a 3D domain with a layered permeability field, define wells and solve a simple production-injection schedule. We begin by loading the `Jutul` package that contains generic features like grids and linear solvers and the `JutulDarcy` package itself. ## Preliminaries

```
using JutulDarcy, Jutul
```

`JutulDarcy` uses SI units internally. It is therefore convenient to define a few constants at the start of the script to have more managable numbers later on.

```
Darcy, bar, kg, meter, day = si_units(:darcy, :bar, :kilogram, :meter, :day);
```

14.49 Defining a porous medium

We start by defining the static part of our simulation problem – the porous medium itself. ### Defining the grid The first step is to create a grid for our simulation domain. We make a tiny 5 by 5 grid with 4 layers that discretizes a physical domain of 2000 by 1500 by 50 meters.

```
nx = ny = 5
nz = 4
dims = (nx, ny, nz)
g = CartesianMesh(dims, (2000.0, 1500.0, 50.0))
#-
```

14.49.1 Adding properties and making a domain

The grid by itself does not fully specify a porous medium. For that we need to specify the permeability in each cell and the porosity. Permeability, often denoted by a positive-definite tensor K, describes the relationship between a pressure gradient and the flow through the medium. Porosity is a dimensionless number between 0 and 1 that describes how much of the porous medium is void space where fluids can be present. The assumption of Darcy flow becomes less reasonable for high porosity values and the flow equations break down at zero porosity. A porosity of 0.2 is then a safe choice. Jutul uses the `DataDomain` type to store a domain/grid together with data. For porous media simulation, `JutulDarcy` includes a convenience function `reservoir_domain` that contains defaults for permeability and porosity. We specify the permeability per-cell with varying values per layer in the vertical direction and a single porosity value for all cells that the function will expand for us. From the output, we can see that basic geometry primitives are also automatically added:

```
nlayer = nx*ny # Cells in each layer
K = vcat(
    fill(0.65, nlayer),
    fill(0.3, nlayer),
    fill(0.5, nlayer),
    fill(0.2, nlayer)
    )*Darcy

domain = reservoir_domain(g, permeability = K, porosity = 0.2)
```

14.50 Defining wells

Now that we have a porous medium with all static properties set up, it is time to introduce some driving forces. Jutul assumes no-flow boundary conditions on all boundary faces unless otherwise specified so we can go ahead and add wells to the model. ### A vertical producer well We will define two wells: A first well is named "Producer" and is a vertical well positioned at `(1, 1)`. By default, the setup_vertical_well function perforates all layers in the model.

```
Prod = setup_vertical_well(domain, 1, 1, name = :Producer);
```

14.50.1 A single-perforation injector

We also define an injector by setup_well. This function allows us to pass a vector of either cell indices or tuples of logical indices that the well trajectory will follow. We setup the injector in the upper left corner.

```
Inj = setup_well(domain, [(nx, ny, 1)], name = :Injector);
```

14.51 Choosing a fluid system

To solve multiphase flow with our little toy reservoir we need to pick a fluid system. The type of system determines what physical effects are modelled, what parameters are required and the runtime and accuracy of the resulting simulation. The choice is in practice a trade-off between accuracy, runtime and available data that should be informed by modelling objectives. In this example our goal is to understand how to set up a simple well problem and the ImmiscibleSystem requires a minimal amount of input. We define liquid and gas phases and their densities at some reference conditions and instantiate the system.

```
## Set up a two-phase immiscible system and define a density secondary variable
phases = (LiquidPhase(), VaporPhase())
rhoLS = 1000.0
rhoGS = 100.0
rhoS = [rhoLS, rhoGS] .* kg/meter^3
sys = ImmiscibleSystem(phases, reference_densities = rhoS)
```

14.51.1 Creating the model

The same fluid system can be used for both flow inside the wells and the reservoir. JutulDarcy treats wells as first-class citizens and models flow inside the well bore using the same fluid description as the reservoir, with modified equations to account for the non-Darcy velocities. We call the utility function that sets up all of this for us. Note that we pass the extra_out argument to get both the model and the parameters values.

```
model, parameters = setup_reservoir_model(domain, sys, wells = [Inj, Prod], extra_out = true)
model
```

The model is an instance of the MultiModel from `Jutul` where a submodel is defined for the reservoir, each of the wells and the facility that controls both wells. In addition we can see the cross-terms that couple these wells together. If we want to see more details on how either of these are set up, we can display for example the reservoir model.

```
reservoir = model[:Reservoir]
```

We can see that the model contains primary variables, secondary variables (sometimes referred to as properties) and static parameters in addition to the system we already set up. These can be replaced or modified to alter the behavior of the system. ### Replace the density function with our custom version Let us change the definition of phase mass densities for our model. We'd like to model our liquid phase as weakly compressible and the vapor phase with more significant compressibility. A common approach is to define densities ρ_α^s at some reference pressure p_r and use a phase compressibility c_α to extrapolate around that known value.

$\rho_\alpha(p) = \rho_\alpha^s \exp((p - p_r)c_\alpha)$

This is already implement in Jutul and we simply need to instantiate the variable definition:

```
c = [1e-6/bar, 1e-4/bar]
  = ConstantCompressibilityDensities(p_ref = 1*bar, density_ref = rhoS, compressibility = c)
```

Before replacing it in the model. This change will propagate to all submodels that have a definition given for PhaseMassDensities, including the wells. The facility, which does not know about densities, will ignore it.

```
replace_variables!(model, PhaseMassDensities =  );
```

This concludes the setup of the model. ## Set up initial state The model is time-dependent and requires initial conditions. For the immiscible model it is sufficient to specify the reference phase pressure and the saturations for both phases, summed up to one. These can be specified per cell or one for the entire grid. Specifying a single pressure for the entire model is not very realistic, but should be fine for our simple example. The initial conditions will equilibrate themselves from gravity fairly quickly.

```
state0 = setup_reservoir_state(model, Pressure = 150*bar, Saturations = [1.0, 0.0])
```

14.52 Set up report time steps and injection rate

We create a set of time-steps. These are report steps where the solution will be reported, but the simulator itself will do internal subdivision of time steps if these values are too coarse for the solvers. We also define an injection rate of a full pore-volume (at reference conditions) of gas.

```
dt = repeat([30.0]*day, 12*5)
pv = pore_volume(model, parameters)
inj_rate = sum(pv)/sum(dt)
```

14.53 Set up well controls

We then set up a total rate target (positive value for injection) together with a corresponding injection control that specifies the mass fractions of the two components/phases for pure gas injection, with surface density given by the known gas density. The producer operates at a fixed bottom hole pressure. These are given as a `Dict` with keys that correspond to the well names.

```
rate_target = TotalRateTarget(inj_rate)
I_ctrl = InjectorControl(rate_target, [0.0, 1.0], density = rhoGS)
bhp_target = BottomHolePressureTarget(50*bar)
P_ctrl = ProducerControl(bhp_target)
controls = Dict()
controls[:Injector] = I_ctrl
controls[:Producer] = P_ctrl
## Set up the forces
```

Set up forces for the whole model. For this example, all other forces than the well controls are defaulted (amounting to no-flow for the reservoir). Jutul supports either a single set of forces for

the entire simulation, or a vector of equal length to `dt` with varying forces. Reasonable operational limits for wells are also set up by default.

```
forces = setup_reservoir_forces(model, control = controls)
```

14.54 Simulate the model

We are finally ready to simulate the model for the given initial state `state0`, report steps `dt`, `parameters` and forces. As the model is small, barring any compilation time, this should run in about 300 ms.

```
result = simulate_reservoir(state0, model, dt, parameters = parameters, forces = forces);
```

14.54.1 Unpacking the result

The result contains a lot of data. This can be unpacked to get the most typical desired outputs: Well responses, reservoir states and the time they correspond to.

```
wd, states, t = result;
```

We could in fact equally well have written `wd, states, t = simulate_reservoir(...)` to arrive at the same result. ## Plot the producer responses We load a plotting package to plot the wells.

```
using GLMakie
```

14.55 Plot the surface rates at the producer

We observe that the total rate does not vary much, but the composition changes from liquid to gas as the front propagate through the domain and hits the producer well. Gas rates:

```
qg = wd[:Producer][:grat];
```

Total rate:

```
qt = wd[:Producer][:rate];
```

Compute liquid rate and plot:

```
ql = qt - qg
x = t/day
fig = Figure()
ax = Axis(fig[1, 1],
    xlabel = "Time (days)",
    ylabel = "Rate (m³/day)",
    title = "Well production rates"
)
lines!(ax, x, abs.(qg).*day, label = "Gas")
lines!(ax, x, abs.(ql).*day, label = "Liquid")
lines!(ax, x, abs.(qt).*day, label = "Total")
axislegend(position = :rb)
fig
```

14.56 Plot bottom hole pressure of the injector

The pressure builds during injection, until the gas breaks through to the other well.

```
bh = wd[:Injector][:bhp]
fig = Figure()
ax = Axis(fig[1, 1],
    xlabel = "Time (days)",
    ylabel = "Bottom hole pressure (bar)",
    title = "Injector bottom hole pressure"
)
lines!(ax, x, bh./bar)
fig
```

14.57 Plot the well results in the interactive viewer

Note that this will open a new window with the plot.

```
plot_well_results(wd, resolution = (800, 500))
```

14.58 Plot the reservoir and final gas saturation field

```
plot_cell_data(g, states[end][:Saturations][1, :], colormap = :seismic)
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Adding new wells to an existing model Taking an existing reservoir model and modifying it with new wells is a typical operation in reservoir simulation. This example demonstrates how to add new wells to an existing model and a few ways to setup these wells.

The example uses the SPE1 data set as the model. We load the dataset as a `JutulCase` as usual and unpack the parts of the problem from the `case` before simulating the base case.

```
using Jutul, JutulDarcy, GLMakie
spe1_pth = JutulDarcy.GeoEnergyIO.test_input_file_path("SPE1", "SPE1.DATA")
case = setup_case_from_data_file(spe1_pth)
(; model, state0, forces, parameters, dt) = case
ws, states = simulate_reservoir(case)
```

14.59 Replacing the existing wells with new wells with the same name

The simplest way to replace a well is to create a new well with the same name so that the new well replaces the old one. This is done by calling the setup routines and using the same name as the original wells. We place one injector and one producer, using the same names as in the original version of the case.

Here, we make use of the `setup_well` function to create the new wells and use the cell indices directly.

```
reservoir = reservoir_domain(model)
I1 = setup_well(reservoir, 1, name = :INJ)
P1 = setup_well(reservoir, 10, name = :PROD)
```

14.60 Add a new producer as multisegment well

We can also add more wells than originally present by adding new wells with new names. Some care will have to be taken later on to ensure that these wells have valid control and limits. We pick a cell with a IJK index to place this producer.

```
P2 = setup_well(reservoir, (10, 10, 1), name = :PROD_NEW, simple_well = false)
```

14.61 Add a new injector with a trajectory

An alternative to placing wells by cell index is to place them by trajectory. We can define a trajectory as a set of points in 3D space, here represented as a Matrix with three columns where each row represents a point along the trajectory. The trajectory is then used to discretize the well into cells.

```
traj = [
    50.0 3100.0 2500.0;
    56.0 2850.0 2540.0;
    120.0 2680.0 2550.0;
    400.0 2600.0 2565.0
]
I2 = setup_well_from_trajectory(reservoir, traj, name = :INJ_NEW)
```

14.62 Set up the new model

The new model is set up by calling the `setup_reservoir_model` function with the old model as the template. This function will create a new model with the same properties and customizations as the original model, but with the new wells added. The template model takes the place of the `sys` argument seen in other examples.

```
new_model = setup_reservoir_model(reservoir, model, wells = [I1, I2, P1, P2]);
```

14.63 Visualize the new wells and the trajectory

We can see the new wells and the trajectory by plotting the model. Note that the trajectory is connected to cell centers in the numerical model. The coarse resolution of the model makes the trajectory appear jagged when realized along cells.

```
fig = plot_reservoir(new_model, title = "New model", alpha = 0.0, edge_color = :black)
lines!(fig.current_axis[], traj', color = :red)
fig
```

14.64 Setup a new state and forces

The new state and forces are set up in the same way as the original model, using the previous state0 and forces as templates. The new control and limits are duplicated from the old ones, mapping producer controls and limits to producers and injector controls and limits to injectors.

We keep the controls of the wells constant throughout the simulation, but we could also have made a `forces` Vector with one value per step.

```
new_state0 = setup_reservoir_state(new_model, state0)

new_control = Dict()
new_limits = Dict()

facility_forces = forces[1][:Facility]

ictrl = facility_forces.control[:INJ]
ilims = facility_forces.limits[:INJ]

pctrl = facility_forces.control[:PROD]
plims = facility_forces.limits[:PROD]

new_control[:INJ] = ictrl
new_control[:PROD] = pctrl
new_limits[:INJ] = ilims
new_limits[:PROD] = plims

new_control[:INJ_NEW] = ictrl
new_limits[:INJ_NEW] = ilims

new_control[:PROD_NEW] = pctrl
new_limits[:PROD_NEW] = plims

new_forces = setup_reservoir_forces(new_model, control = new_control, limits = new_limits)
```

14.65 Simulate the new case

```
new_ws, new_states = simulate_reservoir(new_state0, new_model, dt, forces = new_forces)
```

14.66 Visualize the results

We can visualize the results of the new model and the old model

```
plot_reservoir(new_model, new_states, title = "New wells", step = 120, key = :Rs)
```

14.67 We can also visualize the original model for comparison

```
plot_reservoir(model, states, title = "Original wells", step = 120, key = :Pressure)
```

14.68 Side by side comparison

```
g = physical_representation(reservoir)
fig = Figure(size = (1200, 600))
ax = Axis3(fig[1, 1], zreversed = true, title = "Original wells")
plt = plot_cell_data!(ax, g, states[120][:Saturations][3, :], colorrange = (0.0, 0.7))
ax = Axis3(fig[1, 2], zreversed = true, title = "New wells")
plot_cell_data!(ax, g, new_states[120][:Saturations][3, :], colorrange = (0.0, 0.7))
Colorbar(fig[1, 3], plt)
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # CO2 injection in saline aquifer with storage inventory This example demonstrates a custom K-value compositional model for the injection of CO2 into a saline aquifer. The physical model for flow of CO2 is a realization of the description in 11th SPE Comparative Solutions Project. Simulation of CO2 can be challenging, and we load the HYPRE package to improve performance.

The model also has an option to run immiscible simulations with otherwise identical PVT behavior. This is often faster to run, but lacks the dissolution model present in the compositional version (i.e. no solubility of CO2 in brine, and no vaporization of water in the vapor phase).

```
use_immiscible = false
using Jutul, JutulDarcy
using HYPRE
using GLMakie
nx = 100
nz = 50
Darcy, bar, kg, meter, day, yr = si_units(:darcy, :bar, :kilogram, :meter, :day, :year)
```

14.69 Set up a 2D aquifer model

We set up a Cartesian mesh that is then transformed into an unstructured mesh. We can then modify the coordinates to create a domain with a undulating top surface. CO2 will flow along the top surface and the topography of the top surface has a large impact on where the CO2 migrates.

```
cart_dims = (nx, 1, nz)
physical_dims = (1000.0, 1.0, 50.0)
cart_mesh = CartesianMesh(cart_dims, physical_dims)
```

```
mesh = UnstructuredMesh(cart_mesh, z_is_depth = true)

points = mesh.node_points
for (i, pt) in enumerate(points)
    x, y, z = pt
    x_u = 2* *x/1000.0
    w = 0.2
    dz = 0.05*x + 0.05*abs(x - 500.0)+ w*(30*sin(2.0*x_u) + 20*sin(5.0*x_u))
    points[i] = pt + [0, 0, dz]
end;
```

14.70 Find and plot cells intersected by a deviated injector well

We place a single injector well. This well was unfortunately not drilled completely straight, so we cannot directly use `add_vertical_well` based on logical indices. We instead define a matrix with three columns x, y, z that lie on the well trajectory and use utilities from `Jutul` to find the cells intersected by the trajectory.

```
import Jutul: find_enclosing_cells, plot_mesh_edges
trajectory = [
    645.0 0.5 75;    # First point
    660.0 0.5 85;    # Second point
    710.0 0.5 100.0  # Third point
]

wc = find_enclosing_cells(mesh, trajectory)

fig, ax, plt = plot_mesh_edges(mesh)
plot_mesh!(ax, mesh, cells = wc, transparency = true, alpha = 0.4)
```

View from the side

```
ax.azimuth[] = 1.5*
ax.elevation[] = 0.0
lines!(ax, trajectory', color = :red)
fig
```

14.71 Define permeability and porosity

We loop over all cells and define three layered regions by the K index of each cell. We can then set a corresponding diagonal permeability tensor (3 values) and porosity (scalar) to introduce variation between the layers.

```
nc = number_of_cells(mesh)
perm = zeros(3, nc)
poro = fill(0.3, nc)
region = zeros(Int, nc)
for cell in 1:nc
```

```
    I, J, K = cell_ijk(mesh, cell)
    if K < 0.3*nz
        reg = 1
        permxy = 0.3*Darcy
        phi = 0.2
    elseif K < 0.7*nz
        reg = 2
        permxy = 1.2*Darcy
        phi = 0.35
    else
        reg = 3
        permxy = 0.1*Darcy
        phi = 0.1
    end
    permz = 0.5*permxy
    perm[1, cell] = perm[2, cell] = permxy
    perm[3, cell] = permz
    poro[cell] = phi
    region[cell] = reg
end

fig, ax, plt = plot_cell_data(mesh, poro)
fig
```

14.72 Set up simulation model

We set up a domain and a single injector. We pass the special :co2brine argument in place of the system to the reservoir model setup routine. This will automatically set up a compositional two-component CO2-H2O model with the appropriate functions for density, viscosity and miscibility.

Note that this model by default is isothermal, but we still need to specify a temperature when setting up the model. This is because the properties of CO2 strongly depend on temperature, even when thermal transport is not solved.

The model also accounts for a constant, reservoir-wide salinity. We input mole fractions of salts in the brine so that the solubilities, densities and viscosities for brine cells are corrected in the property model.

```
domain = reservoir_domain(mesh, permeability = perm, porosity = poro, temperature = convert_to_
Injector = setup_well(domain, wc, name = :Injector, simple_well = true)

if use_immiscible
    physics = :immiscible
else
    physics = :kvalue
end
model = setup_reservoir_model(domain, :co2brine,
    wells = Injector,
```

```
    salt_names = ["NaCl", "KCl", "CaSO4", "CaCl2", "MgSO4", "MgCl2"],
    salt_mole_fractions = [0.01, 0.005, 0.005, 0.001, 0.0002, 1e-5],
    co2_physics = physics
);
```

14.73 Customize model by adding relative permeability with hysteresis

We define three relative permeability functions: kro(so) for the brine/liquid phase and krg(g) for both drainage and imbibition. Here we limit the hysteresis to only the non-wetting gas phase, but either combination of wetting or non-wetting hysteresis is supported.

Note that we import a few utilities from JutulDarcy that are not exported by default since hysteresis falls under advanced functionality.

```
import JutulDarcy: table_to_relperm, add_relperm_parameters!, brooks_corey_relperm
so = range(0, 1, 10)
krog_t = so.^2
krog = PhaseRelativePermeability(so, krog_t, label = :og)
```

Higher resolution for second table:

```
sg = range(0, 1, 50);
```

Evaluate Brooks-Corey to generate tables:

```
tab_krg_drain = brooks_corey_relperm.(sg, n = 2, residual = 0.1)
tab_krg_imb = brooks_corey_relperm.(sg, n = 3, residual = 0.25)

krg_drain  = PhaseRelativePermeability(sg, tab_krg_drain, label = :g)
krg_imb  = PhaseRelativePermeability(sg, tab_krg_imb, label = :g)

fig, ax, plt = lines(sg, tab_krg_drain, label = "krg drainage")
lines!(ax, sg, tab_krg_imb, label = "krg imbibition")
lines!(ax, 1 .- so, krog_t, label = "kro")
axislegend()
fig
## Define a relative permeability variable
```

JutulDarcy uses type instances to define how different variables inside the simulation are evaluated. The `ReservoirRelativePermeabilities` type has support for up to three phases with w, ow, og and g relative permeabilities specified as a function of their respective phases. It also supports saturation regions.

Note: If regions are used, all drainage curves come first followed by equal number of imbibition curves. Since we only have a single (implicit) saturation region, the krg input should have two entries: One for drainage, and one for imbibition.

We also call `add_relperm_parameters` to the model. This makes sure that when hysteresis is enabled, we track maximum saturation for hysteresis in each reservoir cell.

```
import JutulDarcy: KilloughHysteresis, ReservoirRelativePermeabilities
krg = (krg_drain, krg_imb)
H_g = KilloughHysteresis() # Other options: CarlsonHysteresis, JargonHysteresis
relperm = ReservoirRelativePermeabilities(g = krg, og = krog, hysteresis_g = H_g)
replace_variables!(model, RelativePermeabilities = relperm)
add_relperm_parameters!(model);
```

14.74 Define approximate hydrostatic pressure and set up initial state

The initial pressure of the water-filled domain is assumed to be at hydrostatic equilibrium. If we use an immiscible model, we must provide the initial saturations. If we are using a compositional model, we should instead provide the overall mole fractions. Note that since both are fractions, and the CO2 model has correspondence between phase ordering and component ordering (i.e. solves for liquid and vapor, and H2O and CO2), we can use the same input value.

```
nc = number_of_cells(mesh)
p0 = zeros(nc)
depth = domain[:cell_centroids][3, :]
g = Jutul.gravity_constant
@. p0 = 160bar + depth*g*1000.0
fig, ax, plt = plot_cell_data(mesh, p0)
fig
```

Set up initial state and parameters

```
if use_immiscible
    state0 = setup_reservoir_state(model,
        Pressure = p0,
        Saturations = [1.0, 0.0],
    )
else
    state0 = setup_reservoir_state(model,
        Pressure = p0,
        OverallMoleFractions = [1.0, 0.0],
    )
end
parameters = setup_parameters(model)
```

14.75 Find the boundary and apply a constant pressureboundary condition

We find cells on the left and right boundary of the model and set a constant pressure boundary condition to represent a bounding aquifer that retains the initial pressure far away from injection.

```
boundary = Int[]
for cell in 1:nc
```

```
    I, J, K = cell_ijk(mesh, cell)
    if I == 1 || I == nx
        push!(boundary, cell)
    end
end
bc = flow_boundary_condition(boundary, domain, p0[boundary], fractional_flow = [1.0, 0.0])
println("Boundary condition added to $(length(bc)) cells.")
```

14.76 Plot the model

```
plot_reservoir(model)
```

14.77 Set up schedule

We set up 25 years of injection and 475 years of migration where the well is shut. The density of the injector is set to 630 kg/m^3, which is roughly the density of CO2 at the in-situ conditions.

```
nstep = 25
nstep_shut = 475
dt_inject = fill(365.0day, nstep)
pv = pore_volume(model, parameters)
inj_rate = 0.075*sum(pv)/sum(dt_inject)

rate_target = TotalRateTarget(inj_rate)
I_ctrl = InjectorControl(rate_target, [0.0, 1.0],
    density = 630.0,
)
```

Set up forces for use in injection

```
controls = Dict(:Injector => I_ctrl)
forces_inject = setup_reservoir_forces(model, control = controls, bc = bc)
```

Forces with shut wells

```
forces_shut = setup_reservoir_forces(model, bc = bc)
dt_shut = fill(365.0day, nstep_shut);
```

Combine the report steps and forces into vectors of equal length

```
dt = vcat(dt_inject, dt_shut)
forces = vcat(
    fill(forces_inject, nstep),
    fill(forces_shut, nstep_shut)
)
println("$nstep report steps with injection, $nstep_shut report steps with migration.")
```

14.78 Add some more outputs for plotting

```
rmodel = reservoir_model(model)
push!(rmodel.output_variables, :RelativePermeabilities)
push!(rmodel.output_variables, :PhaseViscosities)
```

14.79 Simulate the schedule

We set a maximum internal time-step of 30 days to ensure smooth convergence and reduce numerical diffusion.

```
wd, states, t = simulate_reservoir(state0, model, dt,
    parameters = parameters,
    forces = forces,
    max_timestep = 90day
);
```

14.80 Plot the CO2 mole fraction

We plot the overall CO2 mole fraction. We scale the color range to log10 to account for the fact that the mole fraction in cells made up of only the aqueous phase is much smaller than that of cells with only the gaseous phase, where there is almost just CO2.

The aquifer gives some degree of passive flow through the domain, ensuring that much of the dissolved CO2 will leave the reservoir by the end of the injection period.

```
using GLMakie
function plot_co2!(fig, ix, x, title = "")
    ax = Axis3(fig[ix, 1],
        zreversed = true,
        azimuth = -0.51 ,
        elevation = 0.05,
        aspect = (1.0, 1.0, 0.3),
        title = title)
    plt = plot_cell_data!(ax, mesh, x, colormap = :seaborn_icefire_gradient)
    Colorbar(fig[ix, 2], plt)
end
fig = Figure(size = (900, 1200))
for (i, step) in enumerate([5, nstep, nstep + Int(floor(nstep_shut/2)), nstep+nstep_shut])
    if use_immiscible
        plot_co2!(fig, i, states[step][:Saturations][2, :], "CO2 plume saturation at report ste
    else
        plot_co2!(fig, i, log10.(states[step][:OverallMoleFractions][2, :]), "log10 of CO2 mole
    end
end
fig
```

14.81 Plot all relative permeabilities for all time-steps

We can plot all relative permeability evaluations. This both verifies that the hysteresis model is active, but also gives an indication to how many cells are exhibiting imbibition during the simulation.

```
kro_val = Float64[]
krg_val = Float64[]
sg_val = Float64[]
for state in states
    kr_state = state[:RelativePermeabilities]
    s_state = state[:Saturations]
    for c in 1:nc
        push!(kro_val, kr_state[1, c])
        push!(krg_val, kr_state[2, c])
        push!(sg_val, s_state[2, c])
    end
end

fig = Figure()
ax = Axis(fig[1, 1], title = "Relative permeability during simulation")
fig, ax, plt = scatter(sg_val, kro_val, label = "kro", alpha = 0.3)
scatter!(ax, sg_val, krg_val, label = "krg", alpha = 0.3)
axislegend()
fig
```

14.82 Plot result in interactive viewer

If you have interactive plotting available, you can explore the results yourself.

```
plot_reservoir(model, states)
## Calculate and display inventory of CO2
```

We can classify and plot the status of the CO2 in the reservoir. We use a fairly standard classification where CO2 is divided into:

- dissolved CO2 (dissolution trapping)
- residual CO2 (immobile due to zero relative permeability, residual trapping)
- mobile CO2 (mobile but still inside domain)
- outside domain (left the simulation model and migrated outside model)

We also note that some of the mobile CO2 could be considered to be structurally trapped, but this is not classified in our inventory.

```
inventory = co2_inventory(model, wd, states, t)
JutulDarcy.plot_co2_inventory(t, inventory)
```

14.83 Pick a region to investigate the CO2

We can also specify a region to the CO2 inventory. This will introduce additional categories to distinguish between outside and inside the region of interest.

```
cells = findall(region .== 2)
inventory = co2_inventory(model, wd, states, t, cells = cells)
JutulDarcy.plot_co2_inventory(t, inventory)
```

14.84 Define a region of interest using geometry

Another alternative to determine a region of interest is to use geometry. We pick all cells within an ellipsoid a bit away from the injection point.

```
is_inside = fill(false, nc)
centers = domain[:cell_centroids]
for cell in 1:nc
    x, y, z = centers[:, cell]
    is_inside[cell] = sqrt((x - 720.0)^2 + 20*(z-70.0)^2) < 75
end
fig, ax, plt = plot_cell_data(mesh, is_inside)
fig
```

14.85 Plot inventory in ellipsoid

Note that a small mobile dip can be seen when free CO2 passes through this region.

```
inventory = co2_inventory(model, wd, states, t, cells = findall(is_inside))
JutulDarcy.plot_co2_inventory(t, inventory)
```

14.86 Plot the average pressure in the ellipsoid region

Now that we know what cells are within the region of interest, we can easily apply a function over all time-steps to figure out what the average pressure value was.

```
using Statistics
p_avg = map(
    state -> mean(state[:Pressure][is_inside])./bar,
    states
)
lines(t./yr, p_avg,
    axis = (
        title = "Average pressure in region",
        xlabel = "Years", ylabel = "Pressure (bar)"
    )
)
```

14.87 Make a composite plot to correlate CO2 mass in region with spatial distribution

We create a pair of plots that combine both 2D and 3D plots to simultaneously show the ellipsoid, the mass of CO2 in that region for a specific step, and the time series of the CO2 in the same region.

```
stepno = 30
co2_mass_in_region = map(
    state -> sum(state[:TotalMasses][2, is_inside])/1e3,
    states
)
fig = Figure(size = (1200, 600))
ax1 = Axis(fig[1, 1],
    title = "Mass of CO2 in region",
    xlabel = "Years",
    ylabel = "Tonnes CO2"
)
lines!(ax1, t./yr, co2_mass_in_region)
scatter!(ax1, t[stepno]./yr, co2_mass_in_region[stepno], markersize = 12, color = :red)
ax2 = Axis3(fig[1, 2], zreversed = true)
plot_cell_data!(ax2, mesh, states[stepno][:TotalMasses][2, :])
plot_mesh!(ax2, mesh, cells = findall(is_inside), alpha = 0.5)
ax2.azimuth[] = 1.5*
ax2.elevation[] = 0.0
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Hydrostatic equilibriation of models Initializing a reservoir model is an important part of reservoir simulation. For simple, conceptual models it may be sufficient to prescribe a single uniform pressure value as the initial condition, but for more realistic cases the problem of equilibriation becomes very important.

This example gives an overview of how to set up the initial state of a reservoir model using the `setup_reservoir_state` function. To set up a reservoir model, values for all primary variables must be provided for every cell of the domain. The wells are automatically initialized from the cell values. There are two main ways the initial state can be set up:

1. Direct assignment, where the initial values are provided directly as either values for one cell, or values for all cells in the domain. This has the advantage of being very easy to set up, and gives you full control over the initial conditions, but it can be difficult to set up realistic initial that correspond to hydrostatic equilibrium. This means that the model can have significant mass transfer between cells starting from the initial conditions even without wells or boundary conditions.
2. Equilibriation, where the initial values are computed from a set of ordinary differential equations that makes sure the initial state is in hydrostatic equilibrium. This is the most realistic way to set up the initial conditions, but requires additional inputs.

We will go over both of these approaches in this example. The corresponding terminology from typical input files would be keywords like EQUIL (for hydrostatic equilibriation) and PRES, SGAS, SOIL and so on (for direct assignment). If an input file is used, the initial state will set up automatically from these keywords.

```
using Jutul, JutulDarcy, MultiComponentFlash, GLMakie, GeoEnergyIO
```

14.88 Set up a reservoir with depth for the examples

We will use a simple Cartesian mesh with 2 cells in the x-direction and 1000 cells in the z-direction. The domain is 20 m wide, 10 m high and 1000 m deep. Note that we set `z_is_depth = true` to make sure the z coordinate is treated as depth, which is customary for resrvoir models, and assumed by the functions that set up the initial state based on hydrostatic equilibrium. We also shift the mesh so that the top of the domain is at 800 m depth.

```
nx = 2
nz = 1000
Darcy, bar, yr, Kelvin = si_units(:darcy, :bar, :year, :Kelvin)
cmesh = CartesianMesh((nx, 1, nz), (20.0, 10.0, 1000.0))
cmesh = UnstructuredMesh(cmesh, z_is_depth = true)
for (i, pt) in enumerate(cmesh.node_points)
    cmesh.node_points[i] = pt .+ [0.0, 0.0, 800.0]
end
reservoir = reservoir_domain(cmesh, porosity = 0.25, permeability = 0.1*Darcy)
ncells = number_of_cells(cmesh)
```

14.89 Set up a three-phase model with simple initial conditions

We will start by setting up a three-phase model with simple initial conditions. We let the initial conditions be uniform throughout the domain. This is done by passing a scalar for the pressure (which will be repeated for all cells) and a vector with one entry per hphase for the saturations (which will be repeated for all cells into a 3 by ncells matrix).

We also set a typical density function. The density function is important, as it determines the pressure distribution for the equilibrium state.

```
W = AqueousPhase()
O = LiquidPhase()
G = VaporPhase()

rhoS = [1000.0, 700.0, 100.0]
sys_3ph = ImmiscibleSystem((W, O, G), reference_densities = rhoS)
rho = ConstantCompressibilityDensities(sys_3ph, 1.5bar, rhoS, 1e-5/bar)
model_3ph = setup_reservoir_model(reservoir, sys_3ph)
set_secondary_variables!(model_3ph[:Reservoir], PhaseMassDensities = rho)
state0_3ph_1 = setup_reservoir_state(model_3ph, Saturations = [0.0, 1.0, 0.0], Pressure = 200*
```

14.89.1 Simulate the model and plot the change in pressure

We will now simulate the model for 100 years and plot the change in pressure. This could be thought of as the poor man's way of setting up the initial state, as it will give a pressure distribution that is close to hydrostatic equilibrium at the cost of solving many time-steps. For a small model, this cost is negligble, but for larger models it can be significant.

We observe that the initially constant pressure becomes a nearly linear function of depth after 100 years. This is the expected result for a model with constant and no driving forces other than gravity.

```
dt = [convert_to_si(100, :year)]
_, states_3ph_1 = simulate_reservoir(state0_3ph_1, model_3ph, dt)

function plot_comparison(v1, v2, t)
    clim = extrema([v1..., v2...])
    fig = Figure(size = (1000, 500))
    to_mesh(x) = reshape(x, nx, nz)[:, end:-1:1]
    ax1 = Axis(fig[1, 1], title = "$t initial condition")
    heatmap!(ax1, to_mesh(v1), colorrange = clim, colormap = :coolwarm)
    ax2 = Axis(fig[1, 2], title = "$t after 100 years")
    plt = heatmap!(ax2, to_mesh(v2), colorrange = clim, colormap = :coolwarm)
    Colorbar(fig[1, 3], plt)
    fig
end
plot_comparison(state0_3ph_1[:Reservoir][:Pressure]./bar, states_3ph_1[1][:Pressure]./bar, "Pr
```

14.89.2 Confirm that the pressure distribution is approximately linear

The pressure should follow the hydrostatic pressure distribution, i.e. for a single-phase initial state it would be the solution of the ODE:

$\frac{dp}{dz} = \rho(p)g$

Note that as when the variation of the density with respect to pressure is small, this is essentially saying that: $p = \rho g z$

```
function pressure_vs_depth_plot(state)
    if haskey(state, :Reservoir)
        state = state[:Reservoir]
    end
    p = state[:Pressure]./bar
    z = reservoir[:cell_centroids][3, :]
    fig = Figure(size = (300, 600))
    ax = Axis(fig[1, 1], title = "Pressure vs depth", ylabel = "Depth [m]", xlabel = "Pressure
    plt = scatter!(ax, p, z, color = p, markersize = 2, colormap = :coolwarm)
    Colorbar(fig[1, 2], plt)
    fig
end
pressure_vs_depth_plot(states_3ph_1[end])
```

14.89.3 Initialize the model with random initial conditions

We will now set up the model with random initial conditions by providing values per cell. This is done by providing a vector with one entry per cell for pressure, taking care to avoid zero or negative pressures. Similarily, the saturations can be provided as a 3 by ncells matrix, taking care to ensure that the sum of the saturations is equal to one in each cell (i.e. each column of the matrix). This will also return to a hydrostatic distribution, but the saturations make the results more interesting.

```
s0 = 0.2 .+ rand(3, ncells)
s0 = s0 ./ sum(s0, dims = 1)
p0 = 100 .* (0.1 .+ rand(ncells)).*bar
state0_3ph_rand = setup_reservoir_state(model_3ph, Saturations = s0, Pressure = p0)
_, states_3ph_rand = simulate_reservoir(state0_3ph_rand, model_3ph, dt)
plot_comparison(p0./bar, states_3ph_rand[1][:Pressure]./bar, "Pressure [bar]")
```

14.89.4 Plot the water saturation

We will now plot the water saturation for the initial condition and the state after 100 years. We observe that the initially random water saturation migrates to the bottom of the model. This is expected, as the water is the heaviest phase. A similar observation could be made for the other two phases.

```
plot_comparison(s0[1, :], states_3ph_rand[1][:Saturations][1, :], "Water saturation")
```

14.89.5 Plot the pressure distribution for the random initial condition

We now observe a bit more interesting pressure distribution. The slope of the pressure gradient is still linear, but the slope varies in three different regions: One for the top region where gas/vapor migrates, one for the middle region where oil/liquid migrates and one for the bottom region where water migrates.

```
pressure_vs_depth_plot(states_3ph_rand[end])
```

14.90 Equilibriate the model

We will now set up the model with a more realistic initial condition by equilibriating the model. This is done by providing a set of parameters that describe the equilibrium state. The most important parameter is the pressure at a given depth (datum depth and pressure) which fixes the pressure range in the model. In addition, we provide the depths of the fluid contacts where pairs of phases are in contact with each other at hydrostatic equilibrium.

```
eql = EquilibriumRegion(model_3ph, 100*bar, 1000.0, woc = 1600.0, goc = 1100.0)
```

14.90.1 Perform equilibriation

We simply pass the equilibrium region to the `setup_reservoir_state` function as a second positional argument.

```
state0_3_ph_eql = setup_reservoir_state(model_3ph, eql)
```

14.90.2 Plot the results

We plot the initial condition of the pressure (left) and the saturations (right).

```
function plot_state0(state, phases = :wog)
    has_sat = haskey(state[:Reservoir], :Saturations)
    has_comp = haskey(state[:Reservoir], :OverallMoleFractions)
    fig = Figure(size = ((1 + has_sat + has_comp)*300, 500))
    to_mesh(x) = reshape(x, nx, nz)[:, end:-1:1]
    ax1 = Axis(fig[1, 1], title = "Pressure")
    p = state[:Reservoir][:Pressure]./si_unit(:bar)
    plt_pres = heatmap!(ax1, to_mesh(p), colormap = :coolwarm)
    Colorbar(fig[1, 2], plt_pres)

    if has_sat
        sat = state[:Reservoir][:Saturations]
        function to_rgb(i)
            if phases == :wog
                return Makie.RGB(sat[2, i], sat[3, i], sat[1, i])
            elseif phases == :og
                return Makie.RGB(sat[1, i], sat[2, i], 0.0)
            elseif phases == :wg
                return Makie.RGB(0.0, sat[2, i], sat[1, i])
            end
        end
        sat_rgb = map(to_rgb, axes(sat, 2))
        ax2 = Axis(fig[1, 3], title = "Saturation")
        plt = heatmap!(ax2, to_mesh(sat_rgb), colormap = :coolwarm)

        e_wat = PolyElement(color = Makie.RGB(0.0, 0.0, 1.0), strokecolor = :black, strokewidth
        e_oil = PolyElement(color = Makie.RGB(1.0, 0.0, 0.0), strokecolor = :black, strokewidth
        e_gas = PolyElement(color = Makie.RGB(0.0, 1.0, 0.0), strokecolor = :black, strokewidth

        Legend(fig[1, 4],
        [e_wat, e_oil, e_gas],
        ["Aqeous (W)", "Liquid (O)", "Vapor (G)"],
        patchsize = (35, 35), rowgap = 10)
        if has_comp
            comp = state[:Reservoir][:OverallMoleFractions]
            ax3 = Axis(fig[1, 5], title = "Mole fraction, component 1")
            plt_comp = heatmap!(ax3, to_mesh(comp[1, :]), colorrange = (0.0, 1.0))
            Colorbar(fig[1, 6], plt_comp)
        end
    end
    fig
end
plot_state0(state0_3_ph_eql)
```

14.90.3 Plot the pressure distribution by depth

```
pressure_vs_depth_plot(state0_3_ph_eql)
```

14.91 Set up a two-region model

We can also easily set up two different regions in the model. This is done by providing a vector of cell indices for each region, and then passing the vector if regions to the `setup_reservoir_state`. Each region can also be associated with a separate PVT or saturation region.

Note that if the regions are in pressure contact (i.e. there is no flow barrier between them), this state will not be a true equilibrium state and the fluids will move between the regions when simulation starts. The primary utility of setting up multiple regions is to simulate compartments that have different contacts in the same model, for example because a well passes through both compartments.

```
A = Int[]
B = Int[]
for c in 1:ncells
    I, J, K = cell_ijk(cmesh, c)
    if I == 1
        push!(A, c)
    else
        push!(B, c)
    end
end
eql_A = EquilibriumRegion(model_3ph, 100*bar, 1000.0, woc = 1600.0, goc = 1100.0, cells = A)
eql_B = EquilibriumRegion(model_3ph, 120*bar, 1000.0, woc = 1500.0, goc = 1200.0, cells = B)
state0_two_reg = setup_reservoir_state(model_3ph, [eql_A, eql_B])
plot_state0(state0_two_reg)
```

14.92 The two regions are easy to see in a scatter plot

```
pressure_vs_depth_plot(state0_two_reg)
```

14.93 Load a black-oil model with capillary pressure

We load the SPE9 model and transfer the fluid system over to our small reservoir model to see how equilibrium can be computed for a black-oil model.

```
spe9 = GeoEnergyIO.test_input_file_path("SPE9", "SPE9.DATA");
case = setup_case_from_data_file(spe9)
model_bo = setup_reservoir_model(reservoir, case.model);
```

14.93.1 Equilibriate the model

We will now equilibriate the black-oil model. Note that we can specify the Rs value by depth. The density of the liquid phase in this model depends on the dissolved gas. Changing the Rs value will

change the density of the liquid phase, which will in turn change the pressure distribution. Rs can be specified using either rs (constant value) or rs_vs_depth (function of depth). For models with vaporized oil, a similar pair of options is present for Rv.

Note that the oil saturation is not uniform in the oil region, as there is a capillary fringe present. This is a common feature in black-oil models, where significant capillary pressure between the oil and water phases can lead to significant water being "pulled up" into the oil region against the force of gravity until the two effects balance out.

```
eql_bo = EquilibriumRegion(model_bo, 100*bar, 1000.0, woc = 1600.0, goc = 1100.0, rs = 15.0)
state0_bo = setup_reservoir_state(model_bo, eql_bo)
plot_state0(state0_bo)
```

14.94 See the effect of capillary pressure on the pressure distribution

The capillary equilibrium will be accounted. We can see this effect in detail by plotting the capillary pressure and the water saturation by depth. Note that there is connate water in the oil and gas regions, and variation within the oil zone.

```
pc = model_bo[:Reservoir][:CapillaryPressure].pc[1][1] # WO pc, first region
krw = model_bo[:Reservoir][:RelativePermeabilities].krw[1] # WO pc, first region
sw = state0_bo[:Reservoir][:Saturations][1, :]
fig = Figure(size = (1000, 500))
ax1 = Axis(fig[1, 1], title = "Water-Oil capillary pressure", ylabel = "Capillary pressure [ba
plt_pc = lines!(ax1, pc.X, pc.F./bar)
ax2 = Axis(fig[1, 2], title = "Water saturation by depth", ylabel = "Depth [m]", xlabel = "Satu
plt_sw = lines!(ax2, sw, reservoir[:cell_centroids][3, :])
ax3 = Axis(fig[1, 3], title = "Water-Oil relative permeability", ylabel = "Kr", xlabel = "Wate
plt_krw = lines!(ax3, krw.k.X, krw.k.F)
fig
```

14.95 Set up a compositional model

We will now set up a compositional model with two components. The components are methane (will mostly form a gas) and n-decane (will mostly be liquid).

```
c_light = MolecularProperty("Methane")
c_heavy = MolecularProperty("n-Decane")
mixture = MultiComponentMixture([c_light, c_heavy])
eos = GenericCubicEOS(mixture, PengRobinson())
sys_c = MultiPhaseCompositionalSystemLV(eos, (LiquidPhase(), VaporPhase()))
model_comp = setup_reservoir_model(reservoir, sys_c);
```

14.95.1 Equilibriate the model

We will now equilibriate the compositional model. The compositional model can be equilibrated in much the same manner, but unlike the previous model two models, there are additional requirements

on the inputs. Compositional models predict the density and phase saturations based on the composition and temperature and thus specifying composition and temperature as a function of depth is mandatory.

It is important that the compositions are consistent with the phase behavior, i.e. specifying a composition that will end up as a gas for the liquid phase may lead to poor initialization of the model.

Here, we specify the liquid composition as constant and let the vapor composition and temperature be functions of depth.

```
eql_comp = EquilibriumRegion(model_comp, 100*bar, 1000.0,
    goc = 1100.0,
    liquid_composition_vs_depth = z -> [0.0 + 1e-6*z, 1.0 - 1e-6*z],
    vapor_composition = [0.9, 0.1],
    temperature_vs_depth = z -> (300.0 + 30*(z-800.0)/1000.0)*Kelvin
)
state0_comp = setup_reservoir_state(model_comp, eql_comp)
plot_state0(state0_comp, :og)
```

14.96 Set up a single-phase water model

We can also initialize single-phase models, which are entirely parametrized by the contact depth and pressure.

```
sys_1ph = SinglePhaseSystem(reference_density = 1000.0)
model_1ph = setup_reservoir_model(reservoir, sys_1ph)
eql_1ph = EquilibriumRegion(model_1ph, 100*bar, 1000.0)
state0_1ph = setup_reservoir_state(model_1ph, eql_1ph)
plot_state0(state0_1ph)
```

14.97 Set up two-phase water gas

A special case occurs for water-gas systems, where we have to specify the wgc instead of woc/goc. Otherwise, the equilibration is performed in much the same manner.

```
rhoS = [1000.0, 100.0]
sys_wg = ImmiscibleSystem((W, G), reference_densities = rhoS)
rho = ConstantCompressibilityDensities(sys_wg, 1.5bar, rhoS, 1e-5/bar)
model_wg = setup_reservoir_model(reservoir, sys_wg)
set_secondary_variables!(model_wg[:Reservoir], PhaseMassDensities = rho)

eql = EquilibriumRegion(model_wg, 100*bar, 1000.0, wgc = 1300.0)
state0_wg_eql = setup_reservoir_state(model_wg, eql)
plot_state0(state0_wg_eql, :wg)
```

14.98 Conclusion

We have seen how to set up the initial state of a reservoir model using both direct assignment and hydrostatic equilibrium. Even a simple model can have surprising complexity in how the initial state should be set up. We have not covered all the options available (e.g. specifying capillary pressure interfaces), but the examples should give a good idea of how to set up your own model.

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Quarter-five-spot example The quarter-five-spot is a standard test problem that simulates 1/4 of the five spot well pattern by assuming axial symmetry. The problem contains an injector in one corner and the producer in the opposing corner, with a significant volume of fluids injected into the domain.

```
using JutulDarcy, Jutul, HYPRE, Statistics
nx = 50;
```

14.99 Setup

We define a function that, for a given porosity field, computes a solution with an estimated permeability field. For assumptions and derivation of the specific form of the Kozeny-Carman relation used in this example, see Lie, Knut-Andreas. An introduction to reservoir simulation using MATLAB/GNU Octave: User guide for the MATLAB Reservoir Simulation Toolbox (MRST). Cambridge University Press, 2019, Section 2.5.2

```
function perm_kozeny_carman(ϕ)
    return ((ϕ^3)*(1e-5)^2)/(0.81*72*(1-ϕ)^2);
end

function simulate_qfs(porosity = 0.3)
    Dx = 1000.0
    Dz = 10.0
    bar, kg, meter, day = si_units(:bar, :kilogram, :meter, :day)

    mesh = CartesianMesh((nx, nx, 1), (Dx, Dx, Dz))
    K = perm_kozeny_carman.(porosity)
    domain = reservoir_domain(mesh, permeability = K, porosity = porosity)
    Inj = setup_vertical_well(domain, 1, 1, name = :Injector);
    Prod = setup_vertical_well(domain, nx, nx, name = :Producer);
    phases = (LiquidPhase(), VaporPhase())
    rhoLS = 1000.0*kg/meter^3
    rhoGS = 700.0*kg/meter^3
    rhoS = [rhoLS, rhoGS]
    sys = ImmiscibleSystem(phases, reference_densities = rhoS)
    model = setup_reservoir_model(domain, sys, wells = [Inj, Prod])
    c = [1e-6/bar, 1e-6/bar]
      = ConstantCompressibilityDensities(p_ref = 150*bar, density_ref = rhoS, compressibility =
    kr = BrooksCoreyRelativePermeabilities(sys, [2.0, 2.0])
```

```
    replace_variables!(model, PhaseMassDensities =  , RelativePermeabilities = kr);

    state0 = setup_reservoir_state(model, Pressure = 150*bar, Saturations = [1.0, 0.0])
    dt = repeat([30.0]*day, 12*10)
    dt = vcat([0.1, 1.0, 10.0], dt)
    inj_rate = Dx*Dx*Dz*0.3/sum(dt) # 1 PVI if average porosity is 0.3

    rate_target = TotalRateTarget(inj_rate)
    I_ctrl = InjectorControl(rate_target, [0.0, 1.0], density = rhoGS)
    bhp_target = BottomHolePressureTarget(50*bar)
    P_ctrl = ProducerControl(bhp_target)
    controls = Dict()
    controls[:Injector] = I_ctrl
    controls[:Producer] = P_ctrl
    forces = setup_reservoir_forces(model, control = controls)
    return simulate_reservoir(state0, model, dt, forces = forces, info_level = -1)
end
```

14.100 Simulate base case

This will give the solution with uniform porosity of 0.3.

```
ws, states, report_time = simulate_qfs();
```

14.100.1 Plot the solution of the base case

We observe a radial flow pattern initially, before coning occurs near the producer well once the fluid has reached the opposite corner. The uniform permeability and porosity gives axial symmetry at $x = y$.

```
using GLMakie
to_2d(x) = reshape(vec(x), nx, nx)
get_sat(state) = to_2d(state[:Saturations][2, :])
nt = length(report_time)
fig = Figure()
h = nothing
ax = Axis(fig[1, 1])
h = contourf!(ax, get_sat(states[nt÷3]))
ax = Axis(fig[1, 2])
h = contourf!(ax, get_sat(states[nt]))
Colorbar(fig[1, end+1], h)
fig
```

14.101 Create 10 realizations

We create a small set of realizations of the same model, with porosity that is uniformly varying between 0.1 and 0.3. The main idea is to get significantly different flow patterns as the porosity

and permeability changes, and we will return to more realistic porosity fields later in this example.

```
function simulate_porosities(porosities)
    wellsols = []
    s = []
    report_step = Int(ceil(0.5*nt))
    for poro in porosities
        ws_i, states_i, rt = simulate_qfs(poro)
        push!(wellsols, ws_i)
        push!(s, get_sat(states_i[report_step]))
    end
    return (wellsols, s)
end
N = 10
porosities_uniform = []
for i in 1:N
    push!(porosities_uniform, 0.1 .+ 0.2*rand(Float64, (nx*nx)))
end
wells, saturations = simulate_porosities(porosities_uniform);
```

14.101.1 Plot the gas rate at the producer over the ensemble

```
using Statistics
function plot_wells(wellsols)
    fig = Figure(size = (1000, 600))
    ax = Axis(
        fig[1, 1],
        xlabel = "Time [days]",
        ylabel = "Gas rate [m³/s]",
        title = "Producer gas rate",
    )
    t = wellsols[1].time./si_units(:day)
    avg_rate = zeros(length(t))
    for ws in wellsols
        q = abs.(ws[:Producer][:grat])
        avg_rate += q
        lines!(ax, t, q, color = :grey)
    end
    avg_rate ./= length(wellsols)
    lines!(ax, t, avg_rate, color = :red, linewidth = 3, label = "Mean")
    axislegend(ax, position = :lt)
    xlims!(ax, [0.5mean(t), t[end]])
    fig
end

plot_wells(wells)
```

14.101.2 Plot the average saturation over the ensemble

```
avg = mean(saturations)
fig = Figure()
h = nothing
ax = Axis(fig[1, 1])
h = heatmap!(ax, avg, colorrange = (0.0, 1.0))
fig
```

14.101.3 Plot a few realizations of porosity and resulting gas saturation

Note that the porosity fields are uniformly random without any spatial correlation.

```
function plot_realizations(sat, poro)
    fig = Figure(size = (1000, 400))
    poro_crange = (0.15, 0.25)
    sat_crange = (0.5, 1.0)
    h1 = h2 = nothing
    n_to_plot = 5
    for i in 1:n_to_plot
        ax = Axis(fig[1, i], title = "Gas saturation realization $i")
        h1 = heatmap!(ax, sat[i], colorrange = sat_crange)
        ax_poro = Axis(fig[2, i], title = "Porosity realization $i")
        h2 = heatmap!(ax_poro, to_2d(poro[i]), colorrange = poro_crange)
    end
    Colorbar(fig[1, n_to_plot+1], h1)
    Colorbar(fig[2, n_to_plot+1], h2)
    return fig
end
plot_realizations(saturations, porosities_uniform)
```

14.102 Use GeoStats.jl for more realistic porosity fields

Taking uniformly random samples is not a very realistic way to generate porosity fields. A more realistic approach is to use geostatistical methods from GeoStats.jl.

We here use a Gaussian process with a spherical covariance to generate the porosity fields. The setup is taken from the GeoStats.jl documentation which has more details on the approach.

```
import GeoStats: CartesianGrid, GaussianProcess, GaussianVariogram, SphericalCovariance, viz!
N = 30
grid = CartesianGrid(nx, nx)
proc = GaussianProcess(SphericalCovariance(range=30.0), 0.0)
real = rand(proc, grid, N)
```

14.103 Plot mean and variance of the realizations

```
m, v = mean(real), var(real)
fig = Figure(size = (800, 400))
axl = Axis(fig[1, 1], title = "Mean")
axr = Axis(fig[1, 2], title = "Variance")
viz!(axl, m.geometry, color = m.field)
viz!(axr, v.geometry, color = v.field)
fig
```

14.103.1 Run simulations with the new porosity fields

We here map the realizations to porosity values between 0.05 and 0.195, and run the simulations. Note that in a more realistic workflow we would condition the process on data instead of taking unconditional realizations.

```
to_poro(x) = 0.2 + 0.1*clamp(x, -2.0, 2.0)/4.0
porosities_gaussian = map(i -> to_poro.(real[i].field), 1:N)
wells_gaussian, saturations_gaussian = simulate_porosities(porosities_gaussian);
```

14.103.2 Plot the producer rate over the ensemble

```
plot_wells(wells_gaussian)
```

14.103.3 Plot a few realizations for the Gaussian porosity fields

We observe that the porosity fields now have spatial correlation.

```
plot_realizations(saturations_gaussian, porosities_gaussian)
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # A fully differentiable geothermal doublet: History matching and control optimization We are going to set up a conceptual geothermal doublet model in 2D and perform gradient based history matching. This example serves two main purposes: 1. It demonstrates the conceptual workflow for setting up a geothermal model from scratch with a fairly straightforward mesh setup. 2. It shows how to set up a gradient based history matching workflow with the generic optimization interface that allows for optimizing any input parameter used in the setup of a model. ## Load packages and define units

```
using Jutul, JutulDarcy, HYPRE, GLMakie
meter, kilogram, bar, year, liter, second, darcy, day = si_units(:meter, :kilogram, :bar, :yea
```

14.104 Set up the reservoir mesh

The model is a typical geothermal case where there is a layer of high permeability in the middle, confined between two low-permeable layers. For a geothermal model, the low permeable layers are

important, as they store significant amounts of heat that can be conducted to the high permeable layer during production.

We set up the mesh so that the high permeable layer where most of the advective transport occurs has a higher lateral resolution than the low permeable layers. The model is also essentially 2D as there is only one cell thickness in the y direction - a choice that is made to make the example fast to run, especially during the later optimization stages where many simulations must be run to achieve convergence.

```
nx = 50
ntop = 5
nmiddle = 10
nbottom = 5
nz = ntop + nmiddle + nbottom
```

14.104.1 Set up layer thicknesses and vertical cell thicknesses

```
top_layer_thickness = 300.0*meter
middle_layer_thickness = 200.0*meter
bottom_layer_thickness = 300.0*meter
dz = Float64[]
for i in 1:ntop
    push!(dz, top_layer_thickness/ntop)
end
for i in 1:nmiddle
    push!(dz, middle_layer_thickness/nmiddle)
end
for i in 1:ntop
    push!(dz, bottom_layer_thickness/nbottom)
end

cmesh = CartesianMesh((nx, 1, nz), (2000.0, 50.0, dz))
rmesh = UnstructuredMesh(cmesh, z_is_depth = true)
```

14.104.2 Define regions based on our selected depths

We tag each cell with a region number based on its depth. The top layer is region 1, the middle layer is region 2, and the bottom layer is region 3.

```
geo = tpfv_geometry(rmesh)
depths = geo.cell_centroids[3, :]
regions = Int[]
for (i, d_i) in enumerate(depths)
    if d_i <= top_layer_thickness
        r = 1
    elseif d_i <= top_layer_thickness + middle_layer_thickness
        r = 2
    else
        r = 3
```

```
    end
    push!(regions, r)
end
```

14.104.3 Plot the mesh and regions

```
fig, ax, plt = plot_cell_data(rmesh, regions,
    alpha = 0.5,
    outer = true,
    transparency = true,
    colormap = Categorical(:heat)
)
ax.elevation[] = 0.0
ax.azimuth[] =  /2
plot_mesh_edges!(ax, rmesh)
fig
```

14.105 Define functions for setting up the simulation

We will define a function that takes in a Dict with different values and sets up the simulation. The key idea is that we can then optimize the values in the Dict to perform optimization. As we can define any such Dict to set up the model, this interface is very flexible and can be used for both control optimization and history matching with respect to almost any parameter of the model. The disadvantage is that the setup function will be called many times, which can be a substantial cost compared to the more structured optimization interface that only allows for optimization of the numerical parameters (e.g. for the CGNet example). ### Define the time schedule We set up a time schedule for the simulation. The total simulation time is 30 years, and we report the results every 120 days. We also define ten different intervals in this 30 year period, which are the period where we will allow the rates and temperatures to vary during the last part of the optimization tutorial.

```
total_time = 30.0*year
report_step_length = 120.0*day
dt = fill(report_step_length, Int(ceil(total_time/report_step_length)))
num_intervals = 10
interval_interval = total_time/num_intervals
interval_for_step = map(t -> min(Int(ceil(t/interval_interval)), num_intervals), cumsum(dt))
```

14.105.1 Define the wells

We set up two wells, one injector and one producer. The injector is located at the left side of the model, and the producer is located at the right side. We use multisegment wells.

```
base_rate = 15*liter/second
base_temp = 15.0

domain = reservoir_domain(rmesh)
```

```
inj_well = setup_vertical_well(domain, 5, 1,
    heel = ntop+1,
    toe = ntop+nmiddle,
    name = :Injector,
    simple_well = false
)
prod_well = setup_vertical_well(domain, nx - 5, 1,
    heel = ntop+1,
    toe = ntop+nmiddle,
    name = :Producer,
    simple_well = false
)

model_base = setup_reservoir_model(
    domain, :geothermal,
    wells = [inj_well, prod_well],
);
```

14.105.2 Set up a helper to define the forces for a given rate and temperature

```
function setup_doublet_forces(model, inj_temp, inj_rate)
    T_Kelvin = convert_to_si(inj_temp, :Celsius)
    rate_target = TotalRateTarget(inj_rate)
    ctrl_inj  = InjectorControl(rate_target, [1.0],
        density = 1000.0, temperature = T_Kelvin)

    bhp_target = BottomHolePressureTarget(50*bar)
    ctrl_prod = ProducerControl(bhp_target)

    control = Dict(:Injector => ctrl_inj, :Producer => ctrl_prod)
    return setup_reservoir_forces(model, control = control)
end
```

14.105.3 Define the main setup function

This function sets up the model based on the parameters provided in the Dict. It takes in two arguments: The required parameters in a Dict and an optional step_info argument that can be used to set up the model for a specific time step. The function returns a JutulCase object that can be used to simulate the reservoir. Here, we ignore the `step_info` argument and set up the entire schedule every time. Jutul will then automatically use the correct force based on the time step in the simulation.

```
function setup_doublet_case(prm, step_info = missing)
    model = deepcopy(model_base)
    rdomain = reservoir_domain(model)
    rdomain[:permeability] = prm["layer_perm"][regions]
    rdomain[:porosity] = prm["layer_porosities"][regions]
```

```
    rdomain[:rock_heat_capacity] = prm["layer_heat_capacity"][regions]

    T0 = convert_to_si(70, :Celsius)
    thermal_gradient = 20.0/1000.0*meter
    eql = EquilibriumRegion(model, 50*bar, 0.0, temperature_vs_depth = z -> T0 + z*thermal_grad
    state0 = setup_reservoir_state(model, eql)

    forces_per_interval = map((T, rate) -> setup_doublet_forces(model, T, rate),
        prm["injection_temperature_C"], prm["injection_rate"])

    forces = forces_per_interval[interval_for_step]

    return JutulCase(model, dt, forces, state0 = state0)
end
```

14.106 Perform a history match

We first set up a truth case that we will use to generate the data for the history match. We define high perm and porosity in the middle layer, and low perm and porosity in the top and bottom layers before simulating the model.

```
prm_truth = Dict(
    "injection_rate" => fill(base_rate, num_intervals),
    "injection_temperature_C" => fill(base_temp, num_intervals),
    "layer_porosities" => [0.1, 0.3, 0.1],
    "layer_perm" => [0.01, 0.8, 0.02].*darcy,
    "layer_heat_capacity" => [500.0, 600.0, 450.0], # Watt / m K
)
case_truth = setup_doublet_case(prm_truth)
ws, states = simulate_reservoir(case_truth)
```

14.106.1 Define a mismatch objective function

The mismatch objective function is defined as the sum of squares difference between the simulated values and the reference values observed in the wells. Note that we only make use of the well data: - The temperature at the producer well - The mass rate at the producer well (since it is controlled on BHP) - The BHP at the injector well (since it is controlled on rate)

We use the `get_1d_interpolator` function to create interpolators for the reference values, since we cannot assume that the simulator will use exactly the same time-steps as the reference values.

```
prod_rate = ws.wells[:Producer][:wrat]
prod_temp = ws.wells[:Producer][:temperature]
inj_bhp = ws.wells[:Injector][:bhp]

prod_temp_by_time = get_1d_interpolator(ws.time, prod_temp)
prod_rate_by_time = get_1d_interpolator(ws.time, prod_rate)
inj_pressure_by_time = get_1d_interpolator(ws.time, inj_bhp)
```

```
import JutulDarcy: compute_well_qoi
function mismatch_objective(m, s, dt, step_info, forces)
    current_time = step_info[:time]
    ## Current values
    T_at_prod = compute_well_qoi(m, s, forces, :Producer, :temperature)
    rate = compute_well_qoi(m, s, forces, :Producer, :wrat)
    bhp = compute_well_qoi(m, s, forces, :Injector, :bhp)
    ## Reference values
    T_at_prod_ref = prod_temp_by_time(current_time)
    rate_ref = prod_rate_by_time(current_time)
    bhp_ref = inj_pressure_by_time(current_time)
    ## Define mismatch by scaling each term
    T_mismatch = (T_at_prod_ref - T_at_prod)
    rate_mismatch = (rate_ref - rate)*1000
    bhp_mismatch = (bhp - bhp_ref)/bar
    return dt * sqrt(T_mismatch^2 + rate_mismatch^2 + bhp_mismatch^2) / total_time
end
```

14.106.2 Pick an initial guess

We set up an initial guess for the parameters that we will optimize. We assume the injection rate and temperature to be known and we set the porosities and permeabilities to uniform values. The heat capacity is given a bit of layering, but still with completely wrong values.

```
prm_guess = Dict(
    "injection_rate" => fill(base_rate, num_intervals),
    "injection_temperature_C" => fill(base_temp, num_intervals),
    "layer_porosities" => [0.2, 0.2, 0.2],
    "layer_perm" => [0.2, 0.2, 0.2].*darcy,
    "layer_heat_capacity" => [400.0, 400.0, 400.0]
)
case_guess = setup_doublet_case(prm_guess)
ws_guess, states_guess = simulate_reservoir(case_guess)
```

14.106.3 Set up the optimization

We define a dictionary optimization problem that will optimize the parameters in the `prm_guess` dictionary. We start by setting up the object itself, which takes in the initial guess `Dict` and the corresponding setup function.

```
opt = JutulDarcy.setup_reservoir_dict_optimization(prm_guess, setup_doublet_case)
```

14.106.4 Define active parameters and their limits

Note that while the parameters get listed, they are all marked as inactive. We need to explicitly make them free/active and specify a range for each parameter before we can optimize them. We use wide absolute limits for each entry.

```
free_optimization_parameter!(opt, "layer_perm", abs_max = 1.5*darcy, abs_min = 0.01*darcy)
free_optimization_parameter!(opt, "layer_heat_capacity", abs_max = 1000.0, abs_min = 400.0)
free_optimization_parameter!(opt, "layer_porosities", abs_max = 0.35, abs_min = 0.05)
```

14.106.5 Call the optimizer

Now that we have freed a few parameters, we can call the optimizer with the objective function. The defaults for the optimizer are fairly reasonable, so we do not tweak the convergence criteria or the maximum number of iterations. Note that by default the optimizer uses LBFGS, but it is also possible to pass other optimizers as a function callable. The default for the optimizer is to minimize the objective function, which is the case for a history match. By passing for example `lbfgs_num = 1, max_it = 50` it is possible to obtain a better match, but this is not necessary for the purpose of this example.

```
prm_opt = JutulDarcy.optimize_reservoir(opt, mismatch_objective, max_it = 50, gradient_scaling
```

14.106.6 Print the optimization overview

If we display the optimization overview, we can see that there are now additional columns indicating the optimized values. Note that while the permeability and porosities are well matched, the heat capacity of the low permeable layers are not very accurate. There is likely not enough data in the production profiles to constrain the heat capacity of the low permeable layers, as there is limited heat siphoned from these layers in the truth case.

```
opt
```

14.106.7 Simulate the optimized case

```
case_opt = setup_doublet_case(prm_opt)
ws_opt, states_opt = simulate_reservoir(case_opt)
```

14.106.7.1 Plot the well responses

We plot the well responses for the producer temperature, producer water rate, and injector bottom hole pressure. These values represent the data used in the objective function. We observe good match, which is consistent with the reduction in the objective function valeu during the optimization.

```
get_wtime(w) = convert_from_si.(w.time, :day)
get_prod_temp(w) = convert_from_si.(w[:Producer, :temperature], :Celsius)
get_prod_rate(w) = -w[:Producer, :wrat]/si_unit(:liter)
get_inj_bhp(w) = convert_from_si.(w[:Injector, :bhp], :bar)

fig = Figure(size = (1200, 800))
ax = Axis(fig[1, 1], title = "Producer temperature", ylabel = "Temperature (°C)", xlabel = "Ti
scatter!(ax, get_wtime(ws), get_prod_temp(ws), label = "Truth")
lines!(ax, get_wtime(ws_guess), get_prod_temp(ws_guess), label = "Initial guess")
lines!(ax, get_wtime(ws_opt), get_prod_temp(ws_opt), label = "Optimized")
axislegend(position = :rc)
```

```
ax = Axis(fig[2, 1], title = "Producer water rate", ylabel = "Liter / s", xlabel = "Time (days)
scatter!(ax, get_wtime(ws), get_prod_rate(ws), label = "Truth")
lines!(ax, get_wtime(ws_guess), get_prod_rate(ws_guess), label = "Initial guess")
lines!(ax, get_wtime(ws_opt), get_prod_rate(ws_opt), label = "Optimized")
axislegend(position = :rc)
ax = Axis(fig[3, 1], title = "Producer bottom hole pressure", ylabel = "Pressure (bar)", xlabel
scatter!(ax, get_wtime(ws), get_inj_bhp(ws), label = "Truth")
lines!(ax, get_wtime(ws_guess), get_inj_bhp(ws_guess), label = "Initial guess")
lines!(ax, get_wtime(ws_opt), get_inj_bhp(ws_opt), label = "Optimized")
axislegend(position = :rc)
fig
```

14.106.7.2 Plot the spatial results

We plot the spatial results for the truth case, the initial guess, and the optimized case. The temperature is plotted in Celsius and we use the same color scale for all steps. Note that in terms of the optimizer itself, this is hidden data: The objective function only matches the well responses. Getting a good match in the spatial distribution of temperature is a side-effect of the physics and parametrization of the model, as different physics or a different parameterization could lead to good match in terms of the objective function, even without good match for the spatial distribution.

```
step = 80
cmap = reverse(to_colormap(:heat))
fig = Figure(size = (1200, 400))
ax = Axis3(fig[1, 1], title = "Truth")
plot_cell_data!(ax, rmesh, states[step][:Temperature] .- 273.15, colorrange = (10.0, 100.0), co
ax.elevation[] = 0.0
ax.azimuth[] = -/2
hidedecorations!(ax)

ax = Axis3(fig[1, 2], title = "Initial guess")
plot_cell_data!(ax, rmesh, states_guess[step][:Temperature] .- 273.15, colorrange = (10.0, 100
ax.elevation[] = 0.0
ax.azimuth[] = -/2
hidedecorations!(ax)

ax = Axis3(fig[1, 3], title = "Optimized")
plt = plot_cell_data!(ax, rmesh, states_opt[step][:Temperature] .- 273.15, colorrange = (10.0,
ax.elevation[] = 0.0
ax.azimuth[] = -/2
hidedecorations!(ax)
Colorbar(fig[2, 1:3], plt, vertical = false)
fig
```

14.107 Set up control optimization

We can also use the same setup to perform control optimization, where we now can take advantage of the per-interval selection of rates and temperatures. Admittely, this problems is fairly simple, so the optimization is more conceptual than realistic: We define a new objective function that uses a fixed cost for the injected water (per degree times rate) and a similar value of produced heat. To make the optimization problem non-trivial, the cost of additional water (or higher temperature water) is significantly higher than the value of produced water with the same temperature.

```
temperature_injection_cost = 20.0
temperature_production_value = 8.0

function optimization_objective(m, s, dt, step_info, forces)
    T_at_prod = convert_from_si(compute_well_qoi(m, s, forces, :Producer, :temperature), :Cels
    T_at_inj = convert_from_si(forces[:Facility].control[:Injector].temperature, :Celsius)

    mass_rate_injector = compute_well_qoi(m, s, forces, :Injector, :mass_rate)
    mass_rate_producer = compute_well_qoi(m, s, forces, :Producer, :mass_rate)

    cost_inj = abs(mass_rate_injector) * T_at_inj * temperature_injection_cost
    value_prod = abs(mass_rate_producer) * T_at_prod * temperature_production_value
    return dt * (value_prod - cost_inj) / total_time
end

opt_ctrl = JutulDarcy.setup_reservoir_dict_optimization(prm_truth, setup_doublet_case)
```

14.107.1 Set optimization to use injection rate and temperature

Note that as these are represented as per-interval values, we could also have passed vectors of equal length as the number of intervals for more fine-grained control over the limits. We specify that the dependencies include the whole case instead of just state0 and parameters since the forces depend on the optimization parameters.

```
free_optimization_parameter!(opt_ctrl, "injection_temperature_C", abs_max = 80.0, abs_min = 10
free_optimization_parameter!(opt_ctrl, "injection_rate", abs_min = 1.0*liter/second, abs_max =
```

14.107.2 Call the optimizer

```
prm_opt_ctrl = JutulDarcy.optimize_reservoir(opt_ctrl, optimization_objective, maximize = true
opt_ctrl
```

14.107.3 Plot the optimized injection rates and temperatures

The optimized injection rates and temperatures are plotted for each interval. The base case is shown in blue, while the optimized case is shown in orange. Note that the optimized case has reduced the injection temperature to the lower limit for all steps, and instead increase the injection rate significantly. The injection rate has a decrease part-way during the simulation, which increases the residence time of the injected water, allowing additional heat to be siphoned from the low permeable layers.

```
fig = Figure(size = (1200, 400))
ax = Axis(fig[1, 1], title = "Optimized injection temperature", ylabel = "Injection temperature
scatter!(ax, prm_truth["injection_temperature_C"], label = "Base case")
scatter!(ax, prm_opt_ctrl["injection_temperature_C"], label = "Optimized case")
axislegend(position = :rc)

ax = Axis(fig[1, 2], title = "Optimized injection rate", ylabel = "Liter/second", xlabel = "In
scatter!(ax, prm_truth["injection_rate"]./(liter/second), label = "Base case")
scatter!(ax, prm_opt_ctrl["injection_rate"]./(liter/second), label = "Optimized case")
axislegend(position = :rc)
fig
```

14.107.4 Simulate the optimized case

```
case_opt_ctrl = setup_doublet_case(prm_opt_ctrl)
ws_opt_ctrl, states_opt_ctrl = simulate_reservoir(case_opt_ctrl)
```

14.107.5 Plot the distribution of temperature with and without optimization

```
step = 80
cmap = reverse(to_colormap(:heat))
fig = Figure(size = (1000, 400))
ax = Axis3(fig[1, 1], title = "Base case")
plot_cell_data!(ax, rmesh, states[step][:Temperature] .- 273.15, colorrange = (10.0, 100.0), co
ax.elevation[] = 0.0
ax.azimuth[] = -/2
hidedecorations!(ax)

ax = Axis3(fig[1, 2], title = "Optimized")
plt = plot_cell_data!(ax, rmesh, states_opt_ctrl[step][:Temperature] .- 273.15, colorrange = (
ax.elevation[] = 0.0
ax.azimuth[] = -/2
hidedecorations!(ax)
Colorbar(fig[2, 1:2], plt, vertical = false)
fig
```

14.107.6 Plot the total thermal energy in the reservoir

The total thermal energy in the reservoir is computed as the sum of the thermal energy in each cell, which is the result of the rock heat capacity, porosity, fluid heat capacity and the temperature in each cell. The optimized strategy significantly decreases the remaining thermal energy in the reservoir, while still producing less cost than the base case according to our objective. The 2D nature of this problem makes it easy to recover a large amount energy, as the majority of the cells are swept by the cold front.

```
total_energy = map(s -> sum(s[:TotalThermalEnergy]), states)
total_energy_opt = map(s -> sum(s[:TotalThermalEnergy]), states_opt_ctrl)
```

```
fig = Figure(size = (1200, 400))
ax = Axis(fig[1, 1], title = "Total thermal energy", ylabel = "Total remaining energy (megajou
t = ws.time ./ si_unit(:day)
lines!(ax, t, total_energy./1e6, label = "Base case")
lines!(ax, t, total_energy_opt./1e6, label = "Optimized case")
axislegend(position = :rc)
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Hybrid simulation with neural network for relative permeability This example demonstrates how to integrate a neural network into JutulDarcy.jl for relative permeability modeling in reservoir simulations. It includes the following steps: 1. Set up a simple reference simulation with a Brooks-Corey relative permeability model 2. Train a neural network to approximate the Brooks-Corey relative permeability model 3. Incorporate the trained network into a simulation model 4. Compare the results of the neural network-based simulation with the reference simulation

This approach showcases the flexibility of JutulDarcy.jl in incorporating machine learning models into conventional reservoir simulation workflows, potentially enabling more accurate and efficient simulations of complex fluid behavior. ## Preliminaries First, let's import the necessary packages. We will use Lux for the neural network model, due to its explicit representation of the model and the ability to use different optimisers, ideal for integration with Jutul. However, Flux.jl would work just as well for this simple example.

```
using JutulDarcy, Jutul, Lux, ADTypes, Zygote, Optimisers, Random, Statistics, GLMakie
```

14.108 Set up the simulation case

We set up a reference simulation case following the Your first JutulDarcy.jl simulation example: - Create a simple Cartesian Mesh - Convert it to a reservoir domain with permeability and porosity - Set up two wells: a vertical injector and a single-perforation producer #### Fluid system and model setup - Use a two-phase immiscible system (liquid and vapor) - Set reference densities: 1000 kg/m³ for liquid, 100 kg/m³ for vapor #### Timesteps and well controls - Set reporting timesteps: every year for 25 years - Producer: fixed bottom-hole pressure - Injector: high gas injection rate (for dramatic visualization) These steps create a basic simulation setup that we'll use to compare against the neural network-based relative permeability model.

```
Darcy, bar, kg, meter, day = si_units(:darcy, :bar, :kilogram, :meter, :day)

function setup_simulation_case()
    nx = ny = 25
    nz = 10
    cart_dims = (nx, ny, nz)
    physical_dims = (1000.0, 1000.0, 100.0).*meter
    g = CartesianMesh(cart_dims, physical_dims)
    domain = reservoir_domain(g, permeability = 0.3Darcy, porosity = 0.2)
```

```
    Injector = setup_vertical_well(domain, 1, 1, name = :Injector)
    Producer = setup_well(domain, (nx, ny, 1), name = :Producer)

    phases = (LiquidPhase(), VaporPhase())
    rhoLS = 1000.0kg/meter^3
    rhoGS = 100.0kg/meter^3
    reference_densities = [rhoLS, rhoGS]
    sys = ImmiscibleSystem(phases, reference_densities = reference_densities)

    nstep = 25
    dt = fill(365.0day, nstep)

    pv = pore_volume(domain)
    inj_rate = 1.5*sum(pv)/sum(dt)
    rate_target = TotalRateTarget(inj_rate)
    bhp_target = BottomHolePressureTarget(100bar)

    I_ctrl = InjectorControl(rate_target, [0.0, 1.0], density = rhoGS)
    P_ctrl = ProducerControl(bhp_target)
    controls = Dict(:Injector => I_ctrl, :Producer => P_ctrl)

    model, parameters = setup_reservoir_model(domain, sys, wells = [Injector, Producer], extra
    forces = setup_reservoir_forces(model, control = controls);

    return model, parameters, forces, sys, dt
end

ref_model, ref_parameters, ref_forces, ref_sys, ref_dt = setup_simulation_case();
```

The simulation model has a set of default secondary variables (properties) that are used to compute the flow equations. We can have a look at the reservoir model to see what the defaults are for the Darcy flow part of the domain:

```
reservoir_model(ref_model)
```

The secondary variables can be swapped out, replaced and new variables can be added with arbitrary functional dependencies thanks to Jutul's flexible setup for automatic differentiation. Let us adjust the defaults by replacing the relative permeabilities with Brooks-Corey functions: ### Brooks-Corey The Brooks-Corey model is a simple model that can be used to generate relative permeabilities. The model is defined in the mobile region as:

$k_{rw} = k_{max,w}\bar{S}_w$

$k_{rg} = k_{max,g}\bar{S}_g$

where $k_{max,w}$ is the maximum relative permeability, $\bar{S}_w$ is the normalized saturation for the water phase,

$\bar{S}_w = \frac{S_w - S_{wi}}{1 - S_{wi} - S_{rg}}$

and, similarly, for the vapor phase:

$\bar{S}_g = \frac{S_g - S_{rg}}{1 - S_{wi} - S_{rg}}$

We use the Brooks Corey function available in JutulDarcy to evaluate the values for a given saturation range. For simplicity, we use the same exponent and residual saturation for both the liquid and vapour phase, such that we only need to train a single model

```
exponent = 2.0
sr_g = sr_w = 0.2
r_tot = sr_g + sr_w;

kr = BrooksCoreyRelativePermeabilities(ref_sys, [exponent, exponent], [sr_w, sr_g])
replace_variables!(ref_model, RelativePermeabilities = kr);
```

14.108.1 Run the reference simulation

We then set up the initial state with constant pressure and liquid-filled reservoir. The inputs (pressure and saturations) must match the model's primary variables. We can now run the simulation (where we first run a warmup step to avoid JIT compilation overhead).

```
ref_state0 = setup_reservoir_state(ref_model,
    Pressure = 120bar,
    Saturations = [1.0, 0.0]
)

simulate_reservoir(ref_state0, ref_model, ref_dt, parameters = ref_parameters, forces = ref_fo
ref_wd, ref_states, ref_t = simulate_reservoir(ref_state0, ref_model, ref_dt, parameters = ref
```

14.109 Training a neural network to compute relative permeability

The next step is to train a neural network to learn the Brooks-Corey relative permeability curve. While using a neural network to learn a simple analytical function is not typically practical, it serves as a good example for integrating machine learning with reservoir simulation. First we must generate some data for training a model to represent the Brooks Corey function.

```
train_samples = 1000
test_samples = 1000

training_sat = collect(range(Float64(0), stop=Float64(1), length=train_samples))
training_sat = reshape(training_sat, 1, :)

rel_perm_analytical = JutulDarcy.brooks_corey_relperm.(training_sat, n = exponent, residual = s
fig = Figure()
ax = Axis(fig[1,1],
    xlabel = "Saturation",
```

```
    ylabel = "Relative Permeability",
    title = "Saturation vs. Relative Permeability",
    xticks = 0:0.25:1,
    yticks = 0:0.25:1
)
lines!(ax, vec(training_sat), vec(rel_perm_analytical), label="Brooks-Corey RelPerm")
axislegend(ax, position = :lt)
fig
```

14.109.1 Define the neural network architecture

Next we define the neural network architecture. The model takes in a saturation value and outputs a relative permeability value. For a batched input, such as the number of cells in the model, the input and output shapes are (1xN_cells). We define our neural network architecture for relative permeability prediction using a multi-layer perceptron (MLP) with the following characteristics: - Input layer: 1 neuron (saturation value) - Three hidden layers: 16 neurons each, with tanh activation - Output layer: 1 neuron with sigmoid activation (relative permeability)

Key design choices: - Use of tanh activation in hidden layers for smooth first derivatives - Sigmoid in the final layer to constrain output to [0, 1] range - Float64 precision to match JutulDarcy's numerical precision

```
BrooksCoreyMLModel = Chain(
    Dense(1 => 16, tanh),
    Dense(16 => 16, tanh),
    Dense(16 => 16, tanh),
    Dense(16 => 1, sigmoid)
)
```

Define training parameters We train the model using the Adam optimizer with a learning rate of 0.0005. For a total of 20 000 epochs. The `optim` object will store the optimiser momentum, etc.

```
epochs = 20000;
lr = 0.0005;
```

Training loop, using the whole data set epochs number of times. We use Adam for the optimiser, set the random seed and initialise the parameters. Lux defaults to float32 precision, so we need to convert the parameters to float64. Lux uses a stateless, explicit representation of the model. It consists of four parts: - model - the model architecture - parameters - the learnable parameters of the model - states - the state of the model, e.g. the hidden states of the recurrent model - optimiser state - the state of the optimiser, e.g. the momentum In addition, we need to define a rule for automatic differentiation. Here we use Zygote.

```
rng = Random.default_rng()
Random.seed!(rng, 42)

function train_model(ml_model, training_sat, rel_perm_analytical, epochs, lr)
    opt = Optimisers.Adam(lr);
    ps, st = Lux.setup(rng, ml_model);
    ps = ps |> f64;
```

```
    tstate = Lux.Training.TrainState(ml_model, ps, st, opt);
    vjp_rule = ADTypes.AutoZygote();
    loss_function = Lux.MSELoss();

    warmup_data = rand(Float64, 1, 1);
    Training.compute_gradients(vjp_rule, loss_function, (warmup_data, warmup_data), tstate)
    @time begin
        losses = []
        for epoch in 1:epochs
            epoch_losses = []
            _, loss, _, tstate = Lux.Training.single_train_step!(vjp_rule, loss_function, (tra
            push!(epoch_losses, loss)
            append!(losses, epoch_losses)
            if epoch % 1000 == 0 || epoch == epochs
                println("Epoch: $(lpad(epoch, 3)) \t Loss: $(round(mean(epoch_losses), sigdigit
            end
        end
    end

    return tstate, losses
end

tstate, losses = train_model(BrooksCoreyMLModel, training_sat, rel_perm_analytical, epochs, lr)
```

The loss function is plotted to show that the model is learning.

```
fig = Figure()
ax = Axis(fig[1,1],
    xlabel = "Iteration",
    ylabel = "Loss",
    title = "Training Loss",
    yscale = log10
)
lines!(ax, losses, label="per batch")
lines!(ax, epochs:epochs:length(losses),
    mean.(Iterators.partition(losses, epochs)),
    label="epoch mean"
)
axislegend()
fig
```

To test the trained model , we generate some test data, different to the training set

```
testing_sat = sort([0.0; rand(Float64, test_samples-2); 1.0])
testing_sat = reshape(testing_sat, 1, :)
```

Next, we calculate the analytical solution and predicted values with the trained model.

```
test_y = JutulDarcy.brooks_corey_relperm.(testing_sat, n = exponent, residual = sr_g, residual_
pred_y = Lux.apply(BrooksCoreyMLModel, testing_sat, tstate.parameters, tstate.states)[1]

fig = Figure()
ax = Axis(fig[1,1],
    xlabel = "Saturation",
    ylabel = "Relative Permeability",
    title = "Saturation vs. Relative Permeability",
    xticks = 0:0.25:1,
    yticks = 0:0.25:1
)
lines!(ax, vec(testing_sat), vec(test_y), label="Brooks-Corey RelPerm")
lines!(ax, vec(testing_sat), vec(pred_y), label="ML model RelPerm")
axislegend(ax, position = :lt)
fig
```

The plot demonstrates that our neural network has successfully learned to approximate the Brooks-Corey relative permeability curve. This close match between the analytical solution and the ML model's predictions indicates that we can use this trained neural network in our simulation model. ## Replacing the relative permeability function with our neural network Now we can replace the relative permeability function with our neural network. We define a new type `MLModelRelativePermeabilities` that wraps our neural network model and implements the `update_kr!` function. This function is called by the simulator to update the relative permeability values for the liquid and vapour phase. A potential benefit of using a neural network, is that we can compute all the cells in parallel, and access to highly optimised GPU acceleration is trivial.

```
struct MLModelRelativePermeabilities{M, P, S} <: JutulDarcy.AbstractRelativePermeabilities
    ML_model::M
    parameters::P
    states::S
    function MLModelRelativePermeabilities(input_ML_model, parameters, states)
        new{typeof(input_ML_model), typeof(parameters), typeof(states)}(input_ML_model, paramet
    end
end

Jutul.@jutul_secondary function update_kr!(kr, kr_def::MLModelRelativePermeabilities, model, Sa
    ML_model = kr_def.ML_model
    ps = kr_def.parameters
    st = kr_def.states
    for ph in axes(kr, 1)
        sat_batch = reshape(Saturations[ph, :], 1, length(Saturations[ph, :]))
        kr_pred, st = Lux.apply(ML_model, sat_batch, ps, st)
        @inbounds kr[ph, :] .= vec(kr_pred)
    end
end
```

Since JutulDarcy uses automatic differentiation, our new relative permeability model needs to be differentiable. This is inherently satisfied by our neural network model, as differentiability is a

core requirement for machine learning models. One thing to note, is that we are using Lux with Zygote.jl for automatic differentiation, while Jutul uses ForwardDiff.jl. This is not a problem, as the gradient of our simple neural network is fully compatible with ForwardDiff.jl, so no middleware is needed for this integration. We can now replace the default relative permeability model with our new neural network-based model.

```
ml_model, ml_parameters, ml_forces, ml_sys, ml_dt = setup_simulation_case()

ml_kr = MLModelRelativePermeabilities(BrooksCoreyMLModel, tstate.parameters, tstate.states)
replace_variables!(ml_model, RelativePermeabilities = ml_kr);
```

We can now inspect the model to see that the relative permeability model has been replaced.

```
reservoir_model(ml_model)
```

14.109.2 Run the simulation

```
ml_state0 = setup_reservoir_state(ml_model,
    Pressure = 120bar,
    Saturations = [1.0, 0.0]
)

simulate_reservoir(ml_state0, ml_model, ml_dt, parameters = ml_parameters, forces = ml_forces,
ml_wd, ml_states, ml_t = simulate_reservoir(ml_state0, ml_model, ml_dt, parameters = ml_paramet
```

14.109.3 Compare results

We can now compare the results of the reference simulation and the simulation with the neural network-based relative permeability model.

```
function plot_comparison(ref_wd, ml_wd, ref_t, ml_t)
    fig = Figure(size = (1200, 800))

    ax1 = Axis(fig[1, 1], title = "Injector BHP", xlabel = "Time (days)", ylabel = "Pressure (
    lines!(ax1, ref_t/day, ref_wd[:Injector, :bhp]./bar, label = "Brooks-Corey")
    lines!(ax1, ml_t/day, ml_wd[:Injector, :bhp]./bar, label = "ML Model")
    axislegend(ax1)

    ax2 = Axis(fig[1, 2], title = "Producer BHP", xlabel = "Time (days)", ylabel = "Pressure (
    lines!(ax2, ref_t/day, ref_wd[:Producer, :bhp]./bar, label = "Brooks-Corey")
    lines!(ax2, ml_t/day, ml_wd[:Producer, :bhp]./bar, label = "ML Model")
    axislegend(ax2)

    ax3 = Axis(fig[2, 1], title = "Producer Liquid Rate", xlabel = "Time (days)", ylabel = "Rat
    lines!(ax3, ref_t/day, abs.(ref_wd[:Producer, :lrat]).*day, label = "Brooks-Corey")
    lines!(ax3, ml_t/day, abs.(ml_wd[:Producer, :lrat]).*day, label = "ML Model")
    axislegend(ax3)
```

```
    ax4 = Axis(fig[2, 2], title = "Producer Gas Rate", xlabel = "Time (days)", ylabel = "Rate
    lines!(ax4, ref_t/day, abs.(ref_wd[:Producer, :grat]).*day, label = "Brooks-Corey")
    lines!(ax4, ml_t/day, abs.(ml_wd[:Producer, :grat]).*day, label = "ML Model")
    axislegend(ax4)

    return fig
end

plot_comparison(ref_wd, ml_wd, ref_t, ml_t)
```

From the plot, we can see that the neural network-based relative permeability model is able to match the reference simulation to an acceptable level. Interactive visualization of the 3D results is also possible if GLMakie is loaded:

```
plot_reservoir(ml_model, ml_states, key = :Saturations, step = 3)
```

This example demonstrates how to integrate a neural network model for relative permeability into a reservoir simulation using JutulDarcy.jl. While we used a simple Brooks-Corey model for demonstration, this approach can be extended to more complex scenarios where analytical models may not be sufficient. ## Bonus: Improving performance with SimpleChains.jl

When inspecting the simulation results, we observe that using the ML model is slower than the analytical model. This is not surprising, since we are comparing a 593 parameters neural network on a CPU with a simple analytical function. Many popular machine learning libraries prioritize optimization for large neural networks and GPU processing, often at the expense of performance for smaller models and CPU-based computations. For instance, these libraries might use memory allocations to achieve more efficient matrix multiplications, which is beneficial when matrix operations dominate the computation time. However, in our scenario with a relatively small network running on a CPU, we can leverage a specialised library to improve performance. SimpleChains.jl is designed specifically for optimising small neural networks on CPUs. It offers significant performance improvements over traditional deep learning frameworks in such scenarios. Advantages of SimpleChains.jl include: 1. Efficient utilisation of CPU resources, including SIMD vectorisation. 2. Minimal memory allocations during forward and backward passes. 3. Compile-time optimisations specifically for small, fixed-size networks. By using SimpleChains.jl, we can potentially reduce the training time and achieve performance closer to that of the analytical model. Fortunately, Lux.jl makes it straightforward to use SimpleChains as a backend. We simply need to convert our model to a SimpleChains model using the `ToSimpleChainsAdaptor`. This allows us to utilise the SimpleChains backend while still using the Lux training API. (Note: Lux also supports Flux.jl models through a similar adaptor approach.)

```
using SimpleChains
adaptor = ToSimpleChainsAdaptor(static(1));

BrooksCoreyMLModel_sc = adaptor(BrooksCoreyMLModel);
```

We can now train the model using the SimpleChains backend, with the Lux training API.

```
tstate_sc, losses_sc = train_model(BrooksCoreyMLModel_sc, training_sat, rel_perm_analytical, ep
```

The training time should be significantly reduced, since SimpleChains is optimised for small net-

works. With the trained model, we can now replace the relative permeability model in the simulation case, and run the simulation.

```
ml_sc_model, ml_sc_parameters, ml_sc_forces, ml_sc_sys, ml_sc_dt = setup_simulation_case()

ml_sc_kr = MLModelRelativePermeabilities(BrooksCoreyMLModel_sc, tstate_sc.parameters, tstate_sc
replace_variables!(ml_sc_model, RelativePermeabilities = ml_sc_kr);

ml_sc_state0 = setup_reservoir_state(ml_sc_model,
    Pressure = 120bar,
    Saturations = [1.0, 0.0]
)

simulate_reservoir(ml_sc_state0, ml_sc_model, ml_sc_dt, parameters = ml_sc_parameters, forces =
ml_sc_wd, ml_sc_states, ml_sc_t = simulate_reservoir(ml_sc_state0, ml_sc_model, ml_sc_dt, para
```

From the simulation results, we should observe a performance improvement when using the SimpleChains model.

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Model coarsening Running a model at full resolution can be computationally expensive. In many cases, it is possible to coarsen the model to reduce the computational cost. This example demonstrates how to coarsen a model using JutulDarcy.jl. The example uses the Egg model, which is a small oil-water model with heterogeneous permeability. The model is coarsened using different methods and partition sizes, and the results are compared to the fine-scale model. The example demonstrates how to coarsen a model and simulate it using JutulDarcy.jl. The example also demonstrates how to compare the results of the coarse-scale model to the fine-scale model.

The example is intended to show the workflow of coarsening a model, and represents a starting point for more advanced techniques like upscaling, coarse-model calibration and history matching. The model is therefore intentionally simple and very coarse for quick simulations, and not necessarily accurate model responses.

```
using Jutul, JutulDarcy, HYPRE
using GLMakie
using GeoEnergyIO
data_dir = GeoEnergyIO.test_input_file_path("EGG")
data_pth = joinpath(data_dir, "EGG.DATA")
fine_case = setup_case_from_data_file(data_pth);
```

14.110 Simulate the base case

We simulate the fine case to get a reference solution to compare against. We also extract the mesh and reservoir for plotting.

```
fine_model = fine_case.model
fine_reservoir = reservoir_domain(fine_model)
```

```
fine_mesh = physical_representation(fine_reservoir)
ws, states = simulate_reservoir(fine_case, info_level = -1);
```

14.111 Coarsen the model and plot partition

We coarsen the model using a partition size of 20x20x2 and the IJK method where the underlying structure of the mesh is used to subdivide the blocks. This function automatically handles inactive cells and disconnected blocks and can therefore also work on more complex models.

We pass a triplet of integers to specify the partition size. This will give an essentially structured partition. Later on, we we will look at graph partitioners.

```
coarse_case = coarsen_reservoir_case(fine_case, (20, 20, 2), method = :ijk)
coarse_model = coarse_case.model
coarse_reservoir = reservoir_domain(coarse_case)
coarse_mesh = physical_representation(coarse_reservoir)

p = coarse_mesh.partition

fig, = plot_cell_data(fine_mesh, p, colormap = :lipariS)
fig
```

14.111.1 Compare fine-scale and coarse-scale permeability

The fine-scale and coarse-scale permeability fields are compared visually. The coarsening uses a static upscaling, where the permeability is harmonically averaged per direction when coarsening. This is a simple method that can be effective enough for many cases.

The fine-scale permeability is shown on the left, and the coarse-scale is shown on the right, with the same color axis.

```
K_f = fine_reservoir[:permeability][1, :]
K_c = coarse_reservoir[:permeability][1, :]

kcaxis = extrema(K_f)

fig = Figure(size = (1200, 500))
axf = Axis3(fig[1, 1], title = "Fine scale permeability", zreversed = true)
plot_cell_data!(axf, fine_mesh, K_f, colorrange = kcaxis, colormap = :turbo)

axc = Axis3(fig[1, 2], title = "Coarse scale permeability", zreversed = true)
plt = plot_cell_data!(axc, coarse_mesh, K_c, colorrange = kcaxis, colormap = :turbo)
Colorbar(fig[1, 3], plt)
fig
```

14.111.2 Simulate the coarse-scale model

The coarse scale model can be simulated just as the fine-scale model was, but the runtime should be significantly reduced down to around a second.

```
@time ws_c, states_c = simulate_reservoir(coarse_case, info_level = -1);
```

14.111.3 Plot and compare the coarse-scale and fine-scale solutions

We plot the pressure field for the fine-scale and coarse-scale models. The model has little pressure variation, but we see that there are substantial differences between our very coarse model and the original fine-scale.

```
using Statistics
wells = JutulDarcy.get_model_wells(fine_model)

p_c = states_c[end][:Pressure]
p_f = states[end][:Pressure]

caxis = extrema([extrema(p_c)..., extrema(p_f)...])

fig = Figure(size = (1200, 500))
axf = Axis3(fig[1, 1], title = "Fine scale", zreversed = true)
plot_cell_data!(axf, fine_mesh, p_f, colorrange = caxis, colormap = :turbo)

for (k, w) in wells
    plot_well!(axf, fine_mesh, w, fontsize = 0)
end

axc = Axis3(fig[1, 2], title = "Coarse scale", zreversed = true)
plt = plot_cell_data!(axc, coarse_mesh, p_c, colorrange = caxis, colormap = :turbo)

for (k, w) in wells
    plot_well!(axc, fine_mesh, w, fontsize = 0)
end
Colorbar(fig[1, 3], plt)
fig
```

14.111.4 Plot and compare the saturation fields

We observe that the saturation fields are quite different between the coarse-scale and fine scale, with the coarse-scale model showing a more uniform saturation field as the leading shock is smeared away.

```
s_c = states_c[end][:Saturations][1, :]
s_f = states[end][:Saturations][1, :]

scaxis = extrema([extrema(s_c)..., extrema(s_f)...])

fig = Figure(size = (1200, 500))
axf = Axis3(fig[1, 1], title = "Fine scale", zreversed = true)
plot_cell_data!(axf, fine_mesh, s_f, colorrange = scaxis, colormap = :turbo)

for (k, w) in wells
```

```
    plot_well!(axf, fine_mesh, w, fontsize = 0)
end

axc = Axis3(fig[1, 2], title = "Coarse scale", zreversed = true)
plt = plot_cell_data!(axc, coarse_mesh, s_c, colorrange = scaxis, colormap = :turbo)

for (k, w) in wells
    plot_well!(axc, fine_mesh, w, fontsize = 0)
end
Colorbar(fig[1, 3], plt)
fig
```

14.111.5 Plot the average field scale pressure evolution

```
fig = Figure()
axf_p = Axis(fig[1, 1], ylabel = "Avg. pressure / bar")
lines!(axf_p, map(x -> mean(x[:Pressure])/1e5, states), label = "Fine")
lines!(axf_p, map(x -> mean(x[:Pressure])/1e5, states_c), label = "Coarse")
axislegend()
fig
```

14.111.6 Plot the wells interactively

We can plot the well results in the interactive viewer using the comparison feature that allows multiple results to be superimposed in the same figure.

```
plot_well_results([ws, ws_c], names = ["Fine", "Coarse"], field = :orat, accumulated = true)
```

14.111.7 Plot field scale measurables over time interactively

The field-scale quantities match fairly well between the coarse-scale and fine-scale models. There is always a trade-off between accuracy and quality in numerical simulations, where the goal is to find the right balance between accuracy in quantities of interest and computational cost.

```
fine_m = reservoir_measurables(fine_case, ws, states)
coarse_m = reservoir_measurables(coarse_case, ws_c, states_c)
m = copy(fine_m)
for (k, v) in pairs(coarse_m)
    if k != :time
        m[Symbol("coarse_$k")] = v
    end
end

plot_reservoir_measurables(m, left = :fopr, right = :coarse_fopr, accumulated = true)
```

14.112 Compare different partitioning methods

We have only so far tested a single partitioning method. We can quickly generate a few other coarse models using different partitioning methods and coarsening values. We highlight that we can also use the centroids instead of the IJK indices to partition, for when the mesh may not have a structured background mesh. In addition, we can call different graph partitioners by passing the desired number of blocks. Here, we call a simple METIS-based transmissibility coarsening, but the code contains options to use other weights and partitioners.

```
partition_variants = [
    (:centroids, (3, 3, 2)),
    (:ijk, (5, 5, 1)),
    (:metis, 10),
    (:metis, 50)
]

fig = Figure(size = (1200, 600))
layout = GridLayout()
fig[1, 1] = layout
rowwidth = Int(floor(length(partition_variants)/2))
for (no, variant) in enumerate(partition_variants)
    if no > rowwidth
        row = 2
        pix = no - rowwidth
    else
        row = 1
        pix = no
    end
    cmethod, cdim = variant
    variant_case = coarsen_reservoir_case(fine_case, cdim, method = cmethod)
    r = reservoir_domain(variant_case)
    m = physical_representation(r)
    p = m.partition

    ax = Axis3(fig, title = "$cmethod - $cdim", azimuth = 0.3, elevation = 1.0, zreversed = tr
    plot_cell_data!(ax, fine_mesh, p, colormap = :lipariS)
    layout[row, 2*(pix-1) + 1] = ax

    _, variant_states = simulate_reservoir(variant_case, info_level = -1)
    pres = variant_states[end][:Pressure]
    axp = Axis3(fig, title = "Pressure", azimuth = 0.3, elevation = 1.0, zreversed = true)
    for (k, w) in wells
        plot_well!(axp, fine_mesh, w, fontsize = 0)
    end
    plot_cell_data!(axp, m, pres, colorrange = caxis, colormap = :turbo)

    layout[row, 2*(pix-1) + 2] = axp
end
```

```
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Gradient-based optimization of net present value (NPV) One usage of a differentiable simulator is control optimization. A counterpart to history matching, control optimization is the process of finding the optimal controls for a given objective and simulation model. In this example, we will optimize the injection rates for a simplified version of the EGG model to maximize the net present value (NPV) of the reservoir. ## Setting up a coarse model We start off by loading the Egg model and coarsening it to a 20x20x3 grid. We limit the optimization to the first 50 timesteps to speed up the optimization process. If you want to run the optimization for all timesteps on the fine model directly, you can remove the slicing of the case and replace the coarse case with the fine case.

```
using Jutul, JutulDarcy, GLMakie, GeoEnergyIO, HYPRE, LBFGSB
data_dir = GeoEnergyIO.test_input_file_path("EGG")
data_pth = joinpath(data_dir, "EGG.DATA")
fine_case = setup_case_from_data_file(data_pth)
fine_case = fine_case[1:50];
coarse_case = coarsen_reservoir_case(fine_case, (20, 20, 3), method = :ijk);
```

14.113 Set up the rate optimization

We use an utility to set up the rate optimization problem. The utility sets up the objective function, constraints, and initial guess for the optimization. The optimization is set up to maximize the NPV of the reservoir. The contribution to the NPV for a given time t_i given in years is defined as: $\text{NPV}_i = \Delta t_i C_i(q_o, q_w)(1+r)^{-t_i}$

Here, r is the discount rate and $C_i(q_o, q_w)$ is the cash flow for the current rates. The cash flow is defined as the price of producing each phase (oil and water, with oil having a positive price and water a negative price) minus the cost of injecting water.

We set the prices and costs (per barrel) as well as a discount rate of 5% per year. The base rate will be used for all injectors initially, which matches the base case for the Egg model. The utility function takes in a `steps` argument that can be used to set how often the rates are allowed to change.

The values used here are arbitrary for the purposes of the example. You are encouraged to play around with the values to see how the outcome changes in the optimization. Generally higher discount rates prioritze immediate income, while lower or zero discount rates prioritize maximizing the oil production over the entire simulation period.

```
ctrl = coarse_case.forces[1][:Facility].control
base_rate = ctrl[:INJECT1].target.value
function optimize_rates(steps; use_box_bfgs = true)
    setup = JutulDarcy.setup_rate_optimization_objective(coarse_case, base_rate,
        max_rate_factor = 10,
        oil_price = 100.0,
        water_price = -10.0,
```

```
        water_cost = 5.0,
        discount_rate = 0.05,
        maximize = use_box_bfgs,
        sim_arg = (
            rtol = 1e-5,
            tol_cnv = 1e-5
        ),
        steps = steps
    )
    if use_box_bfgs
        obj_best, x_best, hist = Jutul.unit_box_bfgs(setup.x0, setup.obj,
            maximize = true,
            lin_eq = setup.lin_eq
        )
        H = hist.val
    else
        lower = zeros(length(setup.x0))
        upper = ones(length(setup.x0))
        results, x_best = lbfgsb(setup.F!, setup.dF!, setup.x0,
            lb=lower,
            ub=upper,
            iprint = 1,
            factr = 1e12,
            maxfun = 20,
            maxiter = 20,
            m = 20
        )
        H = results
    end
    return (setup.case, H, x_best)
end
```

14.114 Optimize the rates

We optimize the rates for two different strategies. The first strategy is to optimize constant rates for the entire period. The second strategy is to optimize the rates for each report step. ### Optimize with a single set of rates

```
case1, hist1, x1 = optimize_rates(:first);
```

14.114.1 Plot rate allocation for the constant case

```
fig = Figure()
ax = Axis(fig[1, 1], xlabel = "Injector number", ylabel = "Rate fraction (of max injection rate
barplot!(ax, x1)
ax.xticks = eachindex(x1)
fig
```

14.114.2 Optimize with varying rates per time-step

```
case2, hist2, x2 = optimize_rates(:each);
```

14.114.3 Plot rate allocation per well

```
allocs = reshape(x2, length(x1), :)
fig = Figure()
ax = Axis(fig[1, 1], xlabel = "Report step", ylabel = "Rate fraction (of max injection rate)")
for i in axes(allocs, 1)
    lines!(ax, allocs[i, :], label = "Injector #$i")
end
axislegend()
fig
```

14.115 Plot the evolution of the NPV

We plot the evolution of the NPV for the two strategies to compare the results. Note that the optimization produces a higher NPV for the varying rates. This is expected, as the optimization can adjust the rates to the changes in the mobility field during the progress of the simulation.

```
fig = Figure()
ax = Axis(fig[1, 1], xlabel = "LBFGS iteration", ylabel = "Net present value (million USD)")
scatter!(ax, 1:length(hist1), hist1./1e6, label = "Constant rates")
scatter!(ax, 1:length(hist2), hist2./1e6, marker = :x, label = "Varying rates")
axislegend(position = :rb)
fig
```

14.116 Simulate the results

We finally simulate all the cases to compare the results.

```
ws0, states0 = simulate_reservoir(coarse_case, info_level = -1)
ws1, states1 = simulate_reservoir(case1, info_level = -1)
ws2, states2 = simulate_reservoir(case2, info_level = -1)
```

14.116.1 Compute measurables and compare

NPV favors early production, as later income/costs have less relative value. We compute the field measurables to be able to compare the oil and water production.

```
f0 = reservoir_measurables(coarse_case.model, ws0)
f1 = reservoir_measurables(case1.model, ws1)
f2 = reservoir_measurables(case2.model, ws2)
```

14.116.2 Plot the cumulative field oil production

The optimized cases produce more oil than the base case.

```
bbl = si_unit(:stb)
fig = Figure()
ax = Axis(fig[1, 1], xlabel = "Time / days", ylabel = "Field oil production (accumulated, barre
t = ws0.time./si_unit(:day)
lines!(ax, t, f0[:fopt].values./bbl, label = "Base case")
lines!(ax, t, f1[:fopt].values./bbl, label = "Constant rates")
lines!(ax, t, f2[:fopt].values./bbl, label = "Varying rates")
axislegend(position = :rb)
fig
```

14.116.3 Plot the cumulative field water production

The optimized cases produce less water than the base case.

```
fig = Figure()
ax = Axis(fig[1, 1], xlabel = "Time / days", ylabel = "Field water production (accumulated, ba
t = ws0.time./si_unit(:day)
lines!(ax, t, f0[:fwpt].values./bbl, label = "Base case")
lines!(ax, t, f1[:fwpt].values./bbl, label = "Constant rates")
lines!(ax, t, f2[:fwpt].values./bbl, label = "Varying rates")
axislegend(position = :rb)
fig
```

14.116.4 Plot the differences in water saturation

The optimized cases have a different water saturation distribution compared to the base case.

```
reservoir = reservoir_domain(coarse_case.model)
g = physical_representation(reservoir)
function plot_diff!(ax, s)
    plt = plot_cell_data!(ax, g, s0 - s1, colorrange = (-0.5, 0.5), colormap = :balance)
    for (w, wd) in get_model_wells(coarse_case.model)
        plot_well!(ax, g, wd, top_factor = 0.8, fontsize = 10)
    end
    ax.elevation[] = 1.03
    ax.azimuth[] = 3.75
    return plt
end

s0 = states0[end][:Saturations][1, :]
s1 = states1[end][:Saturations][1, :]
s2 = states2[end][:Saturations][1, :]
fig = Figure(size = (1600, 800))
ax = Axis3(fig[1, 1], zreversed = true, title = "Water saturation difference, constant rates")
plt = plot_diff!(ax, s1)
ax = Axis3(fig[1, 2], zreversed = true, title = "Water saturation difference, varying rates")
plot_diff!(ax, s2)
Colorbar(fig[2, :], plt, vertical = false)
```

```
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Adding tracers to a flow simulation JutulDarcy supports tracers, which are concentrations that flow in one or more phases. These tracers can be passive, or they can be active and change the properties.

This example demonstrates passive tracers for a simple model with two injectors. We will add a different type of tracer to each injector so that we can show what region of the reservoir is being flooded by each injector. ## Set up mesh and identify the well cells

```
using Jutul, JutulDarcy, GLMakie
Darcy, kg, meter, day, bar = si_units(:darcy, :kg, :meter, :day, :bar)
ny = 20
nx = 2*ny + 1
nz = 1

g = CartesianMesh((nx, ny, nz), (2000.0, 1000.0, 100.0))

c_i = cell_index(g, (nx÷2+1, ny, 1))
c_p1 = cell_index(g, (1, 1, 1))
c_p2 = cell_index(g, (nx, 1, 1))

fig = Figure()
ax = Axis(fig[1, 1])
Jutul.plot_mesh_edges!(ax, g)
plot_mesh!(ax, g, cells = c_i, color = :red)
plot_mesh!(ax, g, cells = [c_p1, c_p2], color = :blue)
fig
```

14.117 Set up reservoir and well

```
reservoir = reservoir_domain(g, permeability = 0.1Darcy, porosity = 0.1)
c_i1 = cell_index(g, (nx÷2+1, ny, 1))

P1 = setup_vertical_well(reservoir, nx÷2+1, ny, name = :P1)
I1 = setup_vertical_well(reservoir, 1, 1, name = :I1)
I2 = setup_vertical_well(reservoir, nx, 1, name = :I2)
```

14.118 Set up the fluid system

We set up a simple two-phase immiscible system.

```
phases = (AqueousPhase(), LiquidPhase())
rhoWS = rhoLS = 1000.0
```

```
rhoS = [rhoWS, rhoLS] .* kg/meter^3
sys = ImmiscibleSystem(phases, reference_densities = rhoS)
```

14.119 Define the tracers

We set up two tracers, one for each injector. The single-phase tracers are in this case associated with the aqueous phase and will only travel with the aqueous phase. The multiphase tracer will travel with both phases.

```
tracer_1 = SinglePhaseTracer(sys, AqueousPhase())
tracer_2 = MultiPhaseTracer(sys)

tracers = [tracer_1, tracer_2]
```

14.120 Set up the schedule and reporting steps

This is similar to other examples, but we also specify the tracer concentration for the injector wells. Wells without tracer concentration specified will be assumed to have zero concentration for all tracers.

JutulDarcy supports time-varying time-steps, so it is possible to have a a tracer active for a specific time period.

```
dt = repeat([30.0]*day, 100)
pv = pore_volume(reservoir)
inj_rate = 0.35*sum(pv)/sum(dt)

rate_target = TotalRateTarget(inj_rate)
I_ctrl1 = InjectorControl(rate_target, [1.0, 0.0], density = rhoWS, tracers = [0.0, 1.0])
I_ctrl2 = InjectorControl(rate_target, [1.0, 0.0], density = rhoWS, tracers = [1.0, 0.0])

bhp_target = BottomHolePressureTarget(50*bar)
P_ctrl = ProducerControl(bhp_target)
controls = Dict()
controls[:I1] = I_ctrl1
controls[:I2] = I_ctrl2
controls[:P1] = P_ctrl
```

14.121 Set up the reservoir model and simulate

We set up the reservoir model and simulate the flow. Note that we must pass the tracers to the setup function for the reservoir model - otherwise the tracers will not be simulated.

```
model = setup_reservoir_model(reservoir, sys,
    wells = [I1, I2, P1],
    tracers = tracers,
)
```

```
forces = setup_reservoir_forces(model, control = controls)
state0 = setup_reservoir_state(model, Pressure = 100*bar, Saturations = [0.0, 1.0])
ws, states = simulate_reservoir(state0, model, dt, info_level = -1, forces = forces);
```

14.122 Plot interactively

```
plot_reservoir(model, states, key = :TracerMasses, step = length(dt))
```

14.123 Plot the final water saturation and tracer concentrations

Note that the fluid front has filled most of the domain. The tracers give us additional information about what water volume comes from a specific injector.

```
tracer_mass = states[end][:TracerMasses]
sw = states[end][:Saturations][1, :]
c1 = tracer_mass[1, :]
c2 = tracer_mass[2, :]
fig = Figure(size = (600, 1200))
ax = Axis(fig[1, 1], title = "Water saturation")
plot_cell_data!(ax, g, sw)
ax = Axis(fig[2, 1], title = "Tracer from injector 1")
plot_cell_data!(ax, g, c1)
ax = Axis(fig[3, 1], title = "Tracer from injector 2")
plot_cell_data!(ax, g, c2)
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Advanced history matching: Regions, blending and parametric models In this example we will explore a few advanced topics of gradient-based history matching, including how to treat regions and match parametric relative permeability functions. It is recommended that you first read the basic examples on both JutulDarcy simulation models with wells, and the examples on history matching before continuing with this example.

To keep the optimization process speedy, the model used in this example is fairly simple, with three producer wells and a small rectangular domain. We define two rock types that will be given different properties and relative permeability curves, and then we will look at how different prior assumptions will impact the quality of the history match.

```
using Jutul, JutulDarcy, GLMakie
import LBFGSB as lb

nx = 20
g = CartesianMesh((nx, nx, 1), (100.0, 100.0, 10.0))
nc = number_of_cells(g)
```

```
reservoir = reservoir_domain(g);
```

14.124 Define the rock types

We define a model with two rock types, with the middle of the domain containing the second rock type. We also set up a simple plotting function to visualize the results in 2D.

```
Darcy, day, kg, meter, barsa = si_units(:darcy, :day, :kg, :meter, :bar)
X = reservoir[:cell_centroids][1, :]
Y = reservoir[:cell_centroids][2, :]
rocktype = zeros(Int, nc)
for i in 1:nc
    if abs(Y[i] - 50.0) < 20
        rocktype[i] = 2
    else
        rocktype[i] = 1
    end
end

perm = fill(0.1Darcy, nc)
perm[rocktype .== 2] .= 1.0Darcy

function plot_2d(v, title = "")
    fig = Figure()
    ax = Axis(fig[1, 1], title = title)
    plt = heatmap!(ax, reshape(v, nx, nx))
    Colorbar(fig[1, 2], plt)
    fig
end

plot_2d(perm./Darcy, "Permeability / Darcy")
```

14.125 Set up wells

We have a single injector (placed in rock type 2) and three producers where two are placed in rock type 1, and one in rock type 2.

```
I = setup_vertical_well(reservoir, 1, ceil(nx/2), name = :I)
P1 = setup_vertical_well(reservoir, nx, 1, name = :P1)
P2 = setup_vertical_well(reservoir, nx, ceil(nx/2), name = :P2)
P3 = setup_vertical_well(reservoir, nx, nx, name = :P3)

wells = [I, P1, P2, P3]

fig = Figure()
ax = Axis3(fig[1, 1], title = "Reservoir model with two regions", zreversed = true)
plot_cell_data!(ax, g, perm)
```

```
for w in wells
    plot_well!(ax, reservoir, w)
end
fig
```

14.126 Set up a simulation model

We are going to set up several different models, starting with the base case. All models use the same PVT description, but will have different variations of the relative permeability parameters that significantly impact the behavior of the system.

```
phases = (AqueousPhase(), LiquidPhase())
rhoWS = 1000.0
rhoLS = 780.0
rhoS = [rhoWS, rhoLS] .* kg/meter^3
sys = ImmiscibleSystem(phases, reference_densities = rhoS)

function setup_model_with_relperm(kr)
    model = setup_reservoir_model(reservoir, sys, extra_out = false, wells = wells)
    rmodel = reservoir_model(model)
    set_secondary_variables!(rmodel, RelativePermeabilities = kr)
    JutulDarcy.add_relperm_parameters!(rmodel)
    prm = setup_parameters(model)
    return (model, prm)
end
```

14.126.1 Set up the parametric relative permeability model

The base case uses a parametric Brooks-Corey model with different coefficients for each rock type. The parametric relative permeability function exposes parameters in the simulator that can be used in gradient-based optimization with the adjoint method. This is a very powerful concept when combined with Jutul's flexible functionality for chaining together differentiable models. For a Brooks-Corey model the parameters are the exponents, the critical saturation and the maximum relative permeability for both the wetting and non-wetting phases. The parametric version allows you to set these values cell-by-cell, but we set them according to the rock type.

```
kr = JutulDarcy.ParametricCoreyRelativePermeabilities()
model_base, prm_base = setup_model_with_relperm(kr);

prm_res = prm_base[:Reservoir]

is1 = findall(rocktype .== 1)
is2 = findall(rocktype .== 2)

prm_res[:NonWettingKrExponent][is1] .= 1.8
prm_res[:NonWettingKrExponent][is2] .= 3.0

prm_res[:WettingKrExponent][is1] .= 1.0
```

```
prm_res[:WettingKrExponent][is2] .= 1.0

prm_res[:WettingCritical][is1] .= 0.3
prm_res[:WettingCritical][is2] .= 0.1

prm_res[:NonWettingCritical][is1] .= 0.1
prm_res[:NonWettingCritical][is2] .= 0.1

prm_res[:WettingKrMax][is1] .= 0.7
prm_res[:WettingKrMax][is2] .= 1.0

prm_res[:NonWettingKrMax][is1] .= 0.8
prm_res[:NonWettingKrMax][is2] .= 1.0

prm_res
```

14.126.2 Simulate the base case

We set up a simple constant rate injection and bottom hole pressure control for the producers. The injection rate is set to a value that will more or less completely flood the domain. The effect of the different relative permeability parameters in the different rock types is clearly visible when the final water saturation is plotted.

```
dt = fill(30day, 50)
total_time = sum(dt)
irate = sum(pore_volume(reservoir)) / total_time

i_ctrl = InjectorControl(TotalRateTarget(irate), [1.0, 0.0], density = rhoWS)
p_ctrl = ProducerControl(BottomHolePressureTarget(50barsa))

ctrls = Dict(
    :I => i_ctrl,
    :P1 => p_ctrl,
    :P2 => p_ctrl,
    :P3 => p_ctrl,
)
forces = setup_reservoir_forces(model_base, control = ctrls)
state0 = setup_reservoir_state(model_base, Pressure = 90barsa, Saturations = [0.0, 1.0])

simulated_base = simulate_reservoir(state0, model_base, dt,
    forces = forces, parameters = prm_base,
    output_substates = true,
    info_level = -1
)
ws, states = simulated_base
step_to_plot = 34
plot_2d(states[step_to_plot][:Saturations][1, :], "Water saturation at the end of the simulati
```

14.127 Set up the history matching problem

We set up an objective function that, for a given result, produces the sum of squares in mismatch for the wells: Bottom hole pressures and phase rates with approprioate weights. As a sanity check, we make sure that the objective is zero when the truth case is passed.

```
states_base = simulated_base.result.states
w = Float64[]
matches = []
wrat = SurfaceWaterRateTarget(1.0)
orat = SurfaceOilRateTarget(1.0)
bhp = BottomHolePressureTarget(50barsa)

push!(matches, bhp)
push!(w, 1.0/(20*si_unit(:bar)))
push!(matches, wrat)
push!(w, (1/80)*day)
push!(matches, orat)
push!(w, (1/50)*day)

wnames = map(x -> physical_representation(x).name, wells)
o_scale = 1.0/(sum(dt)*length(wnames))
mismatch_objective = (model, state, dt, step_info, forces) -> well_mismatch(
    matches,
    wnames,
    model_base,
    states_base,
    model,
    state,
    dt,
    step_info,
    forces,
    weights = w,
    scale = o_scale,
)
@assert Jutul.evaluate_objective(mismatch_objective, model_base, states_base, dt, forces) == 0
```

14.128 Check the objective function for the base case

We set up a new case with the same model and parameter definitions, but with defaulted parameter values for the Brooks-Corey model. This will obviously give a fairly different set of well curves, but we can quickly verify that the simulation produces a different saturation front and objective.

```
prm_untuned = setup_parameters(model_base)
case_untuned = JutulCase(model_base, dt, forces, state0 = state0, parameters = prm_untuned)
simulated_untuned = simulate_reservoir(case_untuned, output_substates = true, info_level = -1)
ws_untuned, states_untuned = simulated_untuned
```

```
obj0 = Jutul.evaluate_objective(mismatch_objective, model_base, simulated_untuned.result.states
@assert obj0 > 0.0
println("Objective function value for untuned model: ", obj0)
```

14.128.1 Set up a plotting function for the saturation match

```
function plot_sat_match(vals, name = "")
    sref = reshape(states[step_to_plot][:Saturations][1, :], nx, nx)
    sbase = reshape(vals[step_to_plot][:Saturations][1, :], nx, nx)
    fig = Figure(size = (1800, 600))
    ax = Axis(fig[1, 1], title = name)
    plt = heatmap!(ax, sbase, colorrange = (0.0, 1.0))
    ax = Axis(fig[1, 2], title = "Truth case")
    plt = heatmap!(ax, sref, colorrange = (0.0, 1.0))
    Colorbar(fig[1, 3], plt)
    ax = Axis(fig[1, 4], title = "Difference")
    plt = heatmap!(ax, sbase - sref, colorrange = (-1.0, 1.0), colormap = :seismic)
    Colorbar(fig[1, 5], plt)
    fig
end

plot_sat_match(states_untuned, "Defaulted parameters")
```

14.129 Set up the first optimization configuration

We start off this study by doing a straightforward gradient-based optimization of the model where we start from the defaulted parameter values and do not make any assumptions about the number of rock types/facies. In other words, every cell can in principle be assigned a different set of relative permeability parameters if it improves the match against the reference.

We set up some wide bounds for the parameters and set up an optimization configuration. The configuration defines what parameters are to be tuned and the range of values they are allowed to take. As we will see later, there are additional settings that can be used to narrow down the problem definition further, but at the moment we make the following assumptions: - We only tune the relative permeabilities, not other reservoir parameters like volumes or transmissibilities. - Wells are not tuned. - The bounds for the Brooks-Corey parameters are given.

```
cfg = optimization_config(model_base, prm_untuned)
kr_exp_bnds = (1.0, 3.0)
kr_crit_bnds = (0.0, 0.45)
kr_max_bnds = (0.2, 1.0)
for (mkey, mcfg) in pairs(cfg)
    for (k, v) in pairs(mcfg)
        if mkey != :Reservoir
            v[:active] = false
        elseif k == :NonWettingKrExponent || k == :WettingKrExponent
            v[:active] = true
```

```
            v[:abs_min], v[:abs_max] = kr_exp_bnds
        elseif k == :WettingCritical || k == :NonWettingCritical
            v[:active] = true
            v[:abs_min], v[:abs_max] = kr_crit_bnds
        elseif k == :WettingKrMax || k == :NonWettingKrMax
            v[:active] = true
            v[:abs_min], v[:abs_max] = kr_max_bnds
        else
            v[:active] = false
        end
    end
end

opt_setup = JutulDarcy.setup_reservoir_parameter_optimization(case_untuned, mismatch_objective
```

14.130 Set up optimization solver and solve the first set of parameters

We use the LBFGSB solver to solve the optimization problem. We set up a fairly long optimization with 100 function evaluations since we know that the model is tiny and fast to solve. The optimization function returns the optimized parameters and prints the progress to screen.

```
function solve_optimization(setup)
    f! = (x) -> setup.F_and_dF!(NaN, nothing, x)
    g! = (dFdx, x) ->  setup.F_and_dF!(NaN, dFdx, x)
    lower = setup.limits.min
    upper = setup.limits.max
    x0 = setup.x0
    results, final_x = lb.lbfgsb(f!, g!, x0, lb=lower, ub=upper,
        iprint = 0,
        factr = 1e7,
        pgtol = 1e-10,
        maxfun = 100,
        maxiter = 60,
        m = 10
    )
    m = setup.data[:case].model
    prm_tuned = deepcopy(setup.data[:case].parameters)
    devectorize_variables!(prm_tuned, m, final_x, setup.data[:mapper], config = setup.data[:co
    return prm_tuned
end

prm_tune1 = solve_optimization(opt_setup)
ws_tune1, states_tune1 = simulate_reservoir(state0, model_base, dt, forces = forces, parameter
```

14.130.1 Setup plotting for the results

We set up a simple plotting function that will plot the results for the three producers. This is a good way to verify that our objective function is actually measuring the match we want to see.

```
function plot_match(new_ws, new_name)
    fig = Figure(size = (1350, 850))
    for (i, name) in enumerate([:P1, :P2, :P3])
        ax = Axis(fig[1, i], title = "WRAT", ylabel = "m^3/day", xlabel = "days")
        lines!(ax, ws.time./day, -ws[name, :wrat].*day, label = "Truth $name")
        lines!(ax, ws_untuned.time./day, -ws_untuned[name, :wrat].*day, label = "Initial $name
        lines!(ax, new_ws.time./day, -new_ws[name, :wrat].*day, label = "$new_name $name", lin
        axislegend(position = :lt)

        ax = Axis(fig[2, i], title = "ORAT", ylabel = "m^3/day", xlabel = "days")
        lines!(ax, ws.time./day, -ws[name, :orat].*day, label = "Truth $name")
        lines!(ax, ws_untuned.time./day, -ws_untuned[name, :orat].*day, label = "Initial $name
        lines!(ax, new_ws.time./day, -new_ws[name, :orat].*day, label = "$new_name $name", lin
        axislegend()
    end
    fig
end

plot_match(ws_tune1, "Gradient-based match")
```

14.130.2 Plot the saturation front

If we look at the saturation front, we can see that the tuned model is quite different than both the truth case and the untuned case. This is a natural consequence of the history matching problem being ill-posed even for simple models, where multiple solutions will match a sparse set of observations like the ones we have from the wells. The tuned model is not a perfect match to the truth case, but it is an improvement over the base case.

```
plot_sat_match(states_tune1, "Gradient-based match 1")
```

14.130.3 Plot one of the tuned parameters

We can also plot the tuned parameters to see how they look. The tuned parameters vary more or less continously, even though the truth case has distinct regions in place. When the optimization process is conditioned on data from the wells, the optimizer will tend to make the most drastic changes in the near-well region, as these values often have the highest initial gradient.

```
plot_2d(prm_tune1[:Reservoir][:NonWettingKrExponent], "Tuned Wetting Kr exponent")
```

14.131 Set up a second optimization

Let us set up a second optimization problem where we instead of letting the cells vary independently, we assume that the cells are lumped into the two rock types and that we know the distribution of the rock types. This is a more constrained optimization problem, and we can expect that there are

fewer solutions to this problem than the previous version.

```
cfg2 = optimization_config(model_base, prm_untuned)

for (mkey, mcfg) in pairs(cfg2)
    for (k, v) in pairs(mcfg)
        if mkey != :Reservoir
            v[:active] = false
        elseif k == :NonWettingKrExponent || k == :WettingKrExponent
            v[:active] = true
            v[:abs_min], v[:abs_max] = kr_exp_bnds
            v[:lumping] = rocktype
        elseif k == :WettingCritical || k == :NonWettingCritical
            v[:active] = true
            v[:abs_min], v[:abs_max] = kr_crit_bnds
            v[:lumping] = rocktype
        elseif k == :WettingKrMax || k == :NonWettingKrMax
            v[:lumping] = rocktype
            v[:active] = true
            v[:abs_min], v[:abs_max] = kr_max_bnds
            v[:lumping] = rocktype
        else
            v[:active] = false
        end
    end
end
opt_setup_lump = JutulDarcy.setup_reservoir_parameter_optimization(case_untuned, mismatch_obje
prm_tune2 = solve_optimization(opt_setup_lump)
```

14.131.1 Plot the results

We again see excellent match against the reference well solutions.

```
ws_tune2, states_tune2 = simulate_reservoir(state0, model_base, dt, forces = forces, parameters
plot_match(ws_tune2, "Gradient-based match (regions as prior)")
```

14.131.2 Plot the saturation front

The saturation front is also a good match to the truth case as the prior information about the rock types constrained the optimization problem substantially.

```
plot_sat_match(states_tune2, "Gradient-based match 2")
```

14.131.3 Print the parameters

We can also print the tuned parameters to see how they look. The tuned parameters are for the most part fairly close to the truth case, but there are still substantial differences as some parameters may not actually impact the objective much, or can be compensated by other parameters. The problem is still not quite unique – but it is much more constrained than the previous case.

```
function print_match(prm)
    rprm = prm[:Reservoir]
    for k in [:NonWettingKrExponent, :WettingKrExponent, :WettingCritical, :NonWettingCritical
        println("Parameter $k")
        for (regno, cell) in enumerate((is1[1], is2[1]))
            ref = prm_base[:Reservoir][k][cell]
            tuned = rprm[k][cell]
            println("Truth: $ref, Tuned: $tuned")
        end
    end
end
print_match(prm_tune2)
```

14.131.4 Set up a blending variable

Let us flip the problem: What if we know that there are two rock types with corresponding relative permeabilities, but we do not exactly know their distribution? Determining what cells have what rock type is a difficult problem and is in principle discrete (a cell is either rock type 1 or rock type 2). We can use a trick from machine learning to make the problem amenable to gradient based optimization, however, by introducing blending functions that continuously vary between the two rock types. This is a common approach in machine learning and is often referred to as "soft" or "fuzzy" clustering.

We can show the sigmoid function that we will use to blend between the two rock types. The function is defined as: $f(x) = \frac{1}{1+e^{-\alpha x}}$ where the parameter adjusts how steeply the function changes between the two regions at x=1.

```
function sigmoid(x,   = 1.0)
    return 1.0 / (1.0 + exp(- *x))
end

x = range(-1, 1, length = 1000)
y1 = sigmoid.(x, 1.0)
y2 = sigmoid.(x, 2.0)
y5 = sigmoid.(x, 5.0)
y10 = sigmoid.(x, 10.0)
y20 = sigmoid.(x, 20.0)

fig = Figure()
ax = Axis(fig[1, 1], title = "Sigmoid Function", xlabel = "x", ylabel = "sigmoid(x)")
lines!(ax, x, y1, linewidth = 2, label = "sigmoid(x)  =1.0")
lines!(ax, x, y2, linewidth = 2, label = "sigmoid(x)  =2.0")
lines!(ax, x, y5, linewidth = 2, label = "sigmoid(x)  =5.0")
lines!(ax, x, y10, linewidth = 2, label = "sigmoid(x)  =10.0")
lines!(ax, x, y20, linewidth = 2, label = "sigmoid(x)  =20.0")
axislegend(position = :lt)
fig
```

14.132 Plot the blending function for a few regions

We can plot the blending function for a few regions. The blending function transitions smoothly between the different regions as the variable goes from region 1 (value 1) to region 5 (value 5). This choice of function is particularly attractive since it is differentiable everywhere and forms a partition of unity, so that a smooth function can be constructed by combining the weights with the corresponding functions:

$f(x) = \sum_{i=1}^{n} w_i(x) f_i(x)$

```
function myblend(x, low_bnd, n_max,   = 20.0)
    w1 = sigmoid(x - low_bnd - 0.5,  )
    w2 = sigmoid(x - low_bnd + 0.5,  )
    if low_bnd == 1
        w = 1.0 - w1
    elseif low_bnd == n_max
        w = w2
    else
        w = w2 - w1
    end
    return w
end

nmax = 5
x = range(1, nmax, length = 1000)

fig = Figure(size = (800, 400))
ax = Axis(fig[1, 1], title = "Blending Function", xlabel = "x", ylabel = "blend(x)")
for i in 1:nmax
    v = myblend.(x, i, nmax)
    lines!(ax, x, v, linewidth = 2, label = "Region $i")
end
ylims!(ax, (0, 1.2))
ax.xticks = 0:nmax+1
axislegend(position = :lt, nbanks = 5)
fig
```

14.133 Set up blending problem

In our example, if we know that there are two relative permeabilities, the above discussion can be simplified to the case:

$k_r = k_{r1}(p) w_1 + k_{r2} w_2(p)$

Here, k_{r1} and k_{r2} are the two relative permeabilities, and w_1 and w_2 are the weights of the blending function with respect to regions 1 and 2 that depends on the continuous parameter p.

The goal, in terms of programming, is then the following: 1. Set up two relative permeability functions with names 1 and 2 that are distinct and uniform throughout the domain (corresponding to parameter sets 1 and 2 in the truth case). 2. Set up a blending parameter together with a

new relative permeability function that combines the two relative permeabilities according to the blending rule described above. 3. Optimize this case for the blending variable itself, leaving all other variables untouched.

Fortunately for us, the blending functionality is already implemented in Jutul. The setup below is primarily verbose because we need to add another relative permeability function with renamed parameters (adding the suffix 2 to the default names) and letting the blending function know that it is supposed to blend the two named fields RelativePermeabilities1 and RelativePermeabilities2 together with one value per phase in each cell based on the added blending parameter.

```
model_blend, = setup_model_with_relperm(kr);
rmodel_blend = reservoir_model(model_blend)

kr2 = JutulDarcy.ParametricCoreyRelativePermeabilities(
    wetting_exponent = :WettingKrExponent2,
    nonwetting_exponent = :NonWettingKrExponent2,
    wetting_critical = :WettingCritical2,
    nonwetting_critical = :NonWettingCritical2,
    wetting_krmax = :WettingKrMax2,
    nonwetting_krmax = :NonWettingKrMax2
)

nph = number_of_phases(sys)
blend_var = Jutul.BlendingVariable((:RelativePermeabilities1, :RelativePermeabilities2), nph)
blend_par = Jutul.BlendingParameter(nph)

set_secondary_variables!(rmodel_blend,
    RelativePermeabilities1 = kr,
    RelativePermeabilities2 = kr2,
    RelativePermeabilities = blend_var
)

set_parameters!(rmodel_blend,
    BlendingParameter = blend_par,
)

JutulDarcy.add_relperm_parameters!(rmodel_blend.parameters, kr2)
```

14.133.1 Set parameters for the two regions

Since we now have two separate relative permeability functions, each with their own parameters, we set one of them to use the truth parameters for the first region and the other to use the truth parameters for the second region.

The tuning is then done by only tuning the blending parameter.

```
prm_blend_outer = setup_parameters(model_blend)
prm_blend = prm_blend_outer[:Reservoir]

prm_blend[:NonWettingKrExponent] .= 1.8
```

```
prm_blend[:NonWettingKrExponent2] .= 3.0

prm_blend[:WettingKrExponent] .= 1.0
prm_blend[:WettingKrExponent2] .= 1.0

prm_blend[:WettingCritical] .= 0.3
prm_blend[:WettingCritical2] .= 0.1

prm_blend[:NonWettingCritical] .= 0.1
prm_blend[:NonWettingCritical2] .= 0.1

prm_blend[:WettingKrMax] .= 0.7
prm_blend[:WettingKrMax2] .= 1.0

prm_blend[:NonWettingKrMax] .= 0.8
prm_blend[:NonWettingKrMax2] .= 1.0


cfg3 = optimization_config(model_blend, prm_blend_outer)

for (mkey, mcfg) in pairs(cfg3)
    for (k, v) in pairs(mcfg)
        if mkey != :Reservoir
            v[:active] = false
        elseif k == :BlendingParameter
            v[:active] = true
            v[:abs_min], v[:abs_max] = (1.0, 2.0)
        else
            v[:active] = false
        end
    end
end
state0_blend = setup_reservoir_state(model_blend, Pressure = 90barsa, Saturations = [0.0, 1.0])
case_lump = JutulCase(model_blend, dt, forces, state0 = state0_blend, parameters = prm_blend_ou

opt_setup_lump = JutulDarcy.setup_reservoir_parameter_optimization(case_lump, mismatch_objectiv
prm_tune3 = solve_optimization(opt_setup_lump)
```

14.133.2 Plot the results

We plot the blending parameter both as a smooth parameter (what the optimizer works with) and the parameter rounded to a integer (which can then be input to a standard simulator that does not support the blending trick).

```
function plot_blending(vals)
    vals = reshape(vals, nx, nx)
    fig = Figure(size = (1800, 600))
    ax = Axis(fig[1, 1], title = "Tuned parameter")
```

```
    plt = heatmap!(ax, vals, colorrange = (1.0, 2.0), colormap = :vik)
    Colorbar(fig[1, 2], plt)
    ax = Axis(fig[1, 3], title = "Effective region")
    heatmap!(ax, round.(vals), colorrange = (1.0, 2.0), colormap = :vik)
    ax = Axis(fig[1, 4], title = "Truth region")
    plt = heatmap!(ax, reshape(rocktype, nx, nx), colorrange = (1.0, 2.0), colormap = Categori
    Colorbar(fig[1, 5], plt)
    fig
end
plot_blending(prm_tune3[:Reservoir][:BlendingParameter])
```

14.133.3 Verify match

We again see good match against the truth case.

```
ws_tune3, states_tune3 = simulate_reservoir(state0_blend, model_blend, dt, forces = forces, pa
plot_match(ws_tune3, "Gradient-based match (blended regions)")
```

14.133.4 Plot the saturation front

```
plot_sat_match(states_tune3, "Gradient-based match 3 (blended regions)")
```

14.133.5 Run simulation with truncated blending parameter

We can also turn the region into a discrete region by rounding the blending function and check if the match is substantially different. As most cells are firmly in one region or the other, we can expect that the match remains good.

```
prm_tune3_trunc = deepcopy(prm_tune3)
prm_tune3_trunc[:Reservoir][:BlendingParameter] = round.(prm_tune3[:Reservoir][:BlendingParamet
ws_tune3_trunc, = simulate_reservoir(state0_blend, model_blend, dt, forces = forces, parameters
plot_match(ws_tune3_trunc, "Gradient-based match (blended, truncated)")
```

14.134 Optimize regions and parameters at the same time

Finally, we can also optimize the regions and the parameters at the same time. This is a weaker assumption than the two preceeding cases, as the only prior information for the relative permeabilities are the bounds for the parameters and the fact that there are two regions. Consequently, we expect a worse match than in the previous case when looking at saturations and regions.

```
cfg4 = optimization_config(model_blend, prm_blend_outer)

equal_lumping = ones(Int, nc)
for (mkey, mcfg) in pairs(cfg4)
    for (k, v) in pairs(mcfg)
        if mkey != :Reservoir
            v[:active] = false
        elseif k == :NonWettingKrExponent || k == :WettingKrExponent || k == :NonWettingKrExpo
```

```
            v[:active] = true
            v[:abs_min], v[:abs_max] = kr_exp_bnds
            v[:lumping] = equal_lumping
        elseif k == :WettingCritical || k == :NonWettingCritical || k == :WettingCritical2 || k
            v[:active] = true
            v[:abs_min], v[:abs_max] = kr_crit_bnds
            v[:lumping] = equal_lumping
        elseif k == :WettingKrMax || k == :NonWettingKrMax || k == :WettingKrMax2 || k == :NonW
            v[:active] = true
            v[:abs_min], v[:abs_max] = kr_max_bnds
            v[:lumping] = equal_lumping
        elseif k == :BlendingParameter
            v[:active] = true
            v[:abs_min], v[:abs_max] = (1.0, 2.0)
        else
            v[:active] = false
        end
    end
end
opt_setup_lump2 = JutulDarcy.setup_reservoir_parameter_optimization(case_lump, mismatch_objecti
prm_tune4 = solve_optimization(opt_setup_lump2)
```

14.134.1 Plot the match

```
ws_tune4, states_tune4 = simulate_reservoir(state0_blend, model_blend, dt, forces = forces, pa
plot_match(ws_tune4, "Gradient-based match")
```

14.134.2 Plot the saturations

```
plot_sat_match(states_tune4, "Gradient-based match 4")
```

14.134.3 Plot the blending parameter

We recover a reasonable two-region distribution even when simultaneously optimizing both the blending parameter and the relative permeability parameters.

```
plot_blending(prm_tune4[:Reservoir][:BlendingParameter])
```

14.135 Conclusion

History matching is a fundamentally ill-posed problem and prior assumptions is a large part of narrowing down the problem. In this example we have seen a few different ways of looking at the same matching problem depending on the prior modeling assumptions. Gradient-based optimization is a powerful tool for history matching when you have access to a differentiable simulator. The flexibility to allow on-the-fly setup of new functional relationships and corresponding parameters in JutulDarcy nicely complements the gradients, allowing us to reframe the problem to match our

prior assumptions.

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # History matching a coarse model - CGNet This example demonstrates how to calibrate a coarse model against results from a fine model. We do this by optimizing the parameters of the coarse model to match the well curves. This is a implementation of the method described in cgnet1. This also serves as a demonstration of how to use the simulator for history matching, as the fine model results can stand in for real field observations.

14.136 Load and simulate Egg base case

We take a subset of the first 60 steps (1350 days) since not much happens after that in terms of well behavior.

```
using Jutul, JutulDarcy, HYPRE, GeoEnergyIO, GLMakie
import LBFGSB as lb

egg_dir = JutulDarcy.GeoEnergyIO.test_input_file_path("EGG")
data_pth = joinpath(egg_dir, "EGG.DATA")

fine_case = setup_case_from_data_file(data_pth)
fine_case = fine_case[1:60]
simulated_fine = simulate_reservoir(fine_case)
plot_reservoir(fine_case, simulated_fine.states, key = :Saturations, step = 60)
```

14.137 Create initial coarse model and simulate

```
coarse_case = JutulDarcy.coarsen_reservoir_case(fine_case, (25, 25, 5), method = :ijk)
simulated_coarse = simulate_reservoir(coarse_case)
plot_reservoir(coarse_case, simulated_coarse.states, key = :Saturations, step = 60)
```

14.138 Setup optimization

We set up the optimization problem by defining the objective function as a sum of squared mismatches for all well observations, for all time-steps. We also define limits for the parameters, and set up the optimization problem.

We also limit the number of function evaluations since this example runs as a part of continuous integration and we want to keep the runtime short.

```
function setup_optimization_cgnet(case_c, case_f, result_f)
    states_f = result_f.result.states
    wells_results, = result_f
    model_c = case_c.model
    state0_c = setup_state(model_c, case_c.state0);
```

```
param_c = setup_parameters(model_c)
forces_c = case_c.forces
dt = case_c.dt
model_f = case_f.model

bhp = JutulDarcy.BottomHolePressureTarget(1.0)
wells = collect(keys(JutulDarcy.get_model_wells(case_f)))

day = si_unit(:day)
wrat_scale = (1/150)*day
orat_scale = (1/80)*day
grat_scale = (1/1000)*day

w = []
matches = []
signs = []
sys = reservoir_model(model_f).system
wrat = SurfaceWaterRateTarget(-1.0)
orat = SurfaceOilRateTarget(-1.0)
grat = SurfaceGasRateTarget(-1.0)

push!(matches, bhp)
push!(w, 1.0/si_unit(:bar))
push!(signs, 1)

for phase in JutulDarcy.get_phases(sys)
    if phase == LiquidPhase()
        push!(matches, orat)
        push!(w, orat_scale)
        push!(signs, -1)

    elseif phase == VaporPhase()
        push!(matches, grat)
        push!(w, grat_scale)
        push!(signs, -1)
    else
        @assert phase == AqueousPhase()
        push!(matches, wrat)
        push!(w, wrat_scale)
        push!(signs, -1)
    end
end
signs = zeros(Int, length(signs))
o_scale = 1.0/(sum(dt)*length(wells))
G = (model_c, state_c, dt, step_info, forces) -> well_mismatch(
    matches,
    wells,
```

```
        model_f,
        states_f,
        model_c,
        state_c,
        dt,
        step_info,
        forces,
        weights = w,
        scale = o_scale,
        signs = signs
    )

    @assert Jutul.evaluate_objective(G, model_f, states_f, dt, case_f.forces) == 0.0
    ##
    cfg = optimization_config(model_c, param_c,
        use_scaling = true,
        rel_min = 0.001,
        rel_max = 1000
    )
    for (k, v) in cfg
        for (ki, vi) in v
            if ki == :FluidVolume
                vi[:active] = k == :Reservoir
            end
            if ki == :ConnateWater
                vi[:active] = false
            end
            if ki in [:TwoPointGravityDifference, :PhaseViscosities, :PerforationGravityDiffere
                vi[:active] = false
            end
            if ki in [:WellIndices, :Transmissibilities]
                vi[:active] = true
                vi[:abs_min] = 0.0
                vi[:abs_max] = 1e-6
            end

        end
    end
    opt_setup = setup_parameter_optimization(model_c, state0_c, param_c, dt, forces_c, G, cfg)
    x0 = opt_setup.x0
    F0 = opt_setup.F!(x0)
    dF0 = opt_setup.dF!(similar(x0), x0)
    println("Initial objective: $F0, gradient norm $(sum(abs, dF0))")
    return opt_setup
end
```

14.139 Define the optimization loop

JutulDarcy can use any optimization package that can work with gradients and limits, here we use the `LBFGSB` package.

```
function optimize_cgnet(opt_setup)
    lower = opt_setup.limits.min
    upper = opt_setup.limits.max
    x0 = opt_setup.x0
    n = length(x0)
    setup = Dict(:lower => lower, :upper => upper, :x0 => copy(x0))

    prt = 1
    f! = (x) -> opt_setup.F_and_dF!(NaN, nothing, x)
    g! = (dFdx, x) -> opt_setup.F_and_dF!(NaN, dFdx, x)
    results, final_x = lb.lbfgsb(f!, g!, x0, lb=lower, ub=upper,
        iprint = prt,
        factr = 1e12,
        maxfun = 20,
        maxiter = 20,
        m = 20
    )
    return (final_x, results, setup)
end
```

14.140 Run the optimization

```
opt_setup = setup_optimization_cgnet(coarse_case, fine_case, simulated_fine);
final_x, results, setup = optimize_cgnet(opt_setup);
```

14.140.1 Transfer the results back

The optimization is generic and works on a single long vector that represents all our active parameters. We can devectorize this vector back into the nested representation used by the model itself and simulate.

```
tuned_case = deepcopy(opt_setup.data[:case])
model_c = coarse_case.model
model_f = fine_case.model
param_c = tuned_case.parameters
data = opt_setup.data
devectorize_variables!(param_c, model_c, final_x, data[:mapper], config = data[:config])


simulated_tuned = simulate_reservoir(tuned_case);
```

14.140.2 Plot the results interactively

```
using GLMakie

wells_f, = simulated_fine
wells_c, = simulated_coarse
wells_t, states_t, time = simulated_tuned

plot_well_results([wells_f, wells_c, wells_t], time, names = ["Fine", "CGNet-initial", "CGNet-t
```

14.140.3 Create a function to compare individual wells

We next compare individual wells to see how the optimization has affected the match between the coarse scale and fine scale. As we can see, we have reasonably good match between the original model with about 18,000 cells and the coarse model with about 3000 cells. Even better match could be possible by adding more coarse blocks, or also optimizing for example the relative permeability parameters for the coarse model.

We plot the water cut and total rate for the production wells, and the bottom hole pressure for the injection wells.

```
function plot_tuned_well(k, kw; lposition = :lt)
    fig = Figure()
    ax = Axis(fig[1, 1], title = "$k", xlabel = "days", ylabel = "$kw")
    t = wells_f.time./si_unit(:day)
    if kw == :wcut
        F = x -> x[k, :wrat]./x[k, :lrat]
    else
        F = x -> abs.(x[k, kw])
    end

    lines!(ax, t, F(wells_f), label = "Fine-scale")
    lines!(ax, t, F(wells_c), label = "Initial coarse")
    lines!(ax, t, F(wells_t), label = "Tuned coarse")
    axislegend(position = lposition)
    fig
end
```

14.140.4 Plot PROD1 water cut

```
plot_tuned_well(:PROD1, :wcut)
```

14.140.5 Plot PROD2 water cut

```
plot_tuned_well(:PROD2, :wcut)
```

14.140.6 Plot PROD4 water cut

```
plot_tuned_well(:PROD4, :wcut)
```

14.140.7 Plot PROD1 total rate

```
plot_tuned_well(:PROD1, :rate)
```

14.140.8 Plot PROD2 total rate

```
plot_tuned_well(:PROD2, :rate)
```

14.140.9 Plot PROD4 total rate

```
plot_tuned_well(:PROD4, :rate)
```

14.140.10 Plot INJECT1 bhp

```
plot_tuned_well(:INJECT1, :bhp, lposition = :rt)
```

14.140.11 Plot INJECT4 bhp

```
plot_tuned_well(:INJECT4, :bhp, lposition = :rt)
```

14.141 Plot the objective evaluations during optimization

```
fig = Figure()
ys = log10
is = x -> x
ax1 = Axis(fig[1, 1], yscale = ys, title = "Objective evaluations", xlabel = "Iterations", ylab
plot!(ax1, opt_setup[:data][:obj_hist][2:end])
fig
```

14.142 Plot the evoluation of scaled parameters

We show the difference between the initial and final values of the scaled parameters, as well as the lower bound.

JutulDarcy maps the parameters to a single vector for optimization with values that are approximately in the box limit range (0, 1). This is convenient for optimizers, but can also be useful when plotting the parameters, even if the units are not preserved in this visualization, only the magnitude.

```
fig = Figure(size = (800, 600))
ax1 = Axis(fig[1, 1], title = "Scaled parameters", ylabel = "Scaled value")
```

```
scatter!(ax1, setup[:x0], label = "Initial X")
scatter!(ax1, final_x, label = "Final X", markersize = 5)
lines!(ax1, setup[:lower], label = "Lower bound")
axislegend()

trans = data[:mapper][:Reservoir][:Transmissibilities]

function plot_bracket(v, k)
    start = v.offset_x+1
    stop = v.offset_x+v.n_x
    y0 = setup[:lower][start]
    y1 = setup[:lower][stop]
    bracket!(ax1, start, y0, stop, y1,
    text = "$k", offset = 1, orientation = :down)
end

for (k, v) in pairs(data[:mapper][:Reservoir])
    plot_bracket(v, k)
end
ylims!(ax1, (-0.2*maximum(final_x), nothing))
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Gradient-based matching of parameters against observations We create a simple test problem: A 1D nonlinear displacement. The observations are generated by solving the same problem with the true parameters. We then match the parameters against the observations using a different starting guess for the parameters, but otherwise the same physical description of the system.

This example uses the numerical parameter optimization framework in Jutul for model calibration. Later on, the high-level interface for optimization is also demonstrated.

```
using Jutul
using JutulDarcy
using LinearAlgebra
using GLMakie
```

14.143 Define a setup function

```
irate = 10.0
function setup_bl(;nc = 100, time = 1.0, nstep = 100, poro = 0.1, perm = 0.1*si_unit(:darcy))
    T = time
    tstep = repeat([T/nstep], nstep)
    G = get_1d_reservoir(nc, poro = poro, perm = perm)
    nc = number_of_cells(G)
```

```
    bar = 1e5
    p0 = 1000*bar
    sys = ImmiscibleSystem((LiquidPhase(), VaporPhase()))
    model = SimulationModel(G, sys)
    model.primary_variables[:Pressure] = Pressure(minimum = -Inf, max_rel = nothing)
    kr = BrooksCoreyRelativePermeabilities(sys, [2.0, 2.0])
    replace_variables!(model, RelativePermeabilities = kr)
    tot_time = sum(tstep)

    parameters = setup_parameters(model, PhaseViscosities = [1e-3, 5e-3]) # 1 and 5 cP
    state0 = setup_state(model, Pressure = p0, Saturations = [0.0, 1.0])

    src  = [SourceTerm(1, irate, fractional_flow = [1.0-1e-3, 1e-3]),
            SourceTerm(nc, -irate, fractional_flow = [1.0, 0.0])]
    forces = setup_forces(model, sources = src)

    return JutulCase(model, tstep, forces, state0 = state0, parameters = parameters)
end
```

Number of cells and time-steps

```
N = 100
Nt = 100
poro_ref = 0.1
perm_ref = 0.1*si_unit(:darcy)
```

14.144 Set up and simulate reference

```
case_ref = setup_bl(nc = N, nstep = Nt, poro = poro_ref, perm = perm_ref)
states_ref, = simulate(case_ref);
```

14.145 Set up another case where the porosity is different

```
case_dporo = setup_bl(nc = N, nstep = Nt, poro = 2*poro_ref, perm = 1.0*perm_ref)
states, rep = simulate(case_dporo);
```

14.146 Plot the results

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Saturation")
lines!(ax, states_ref[end][:Saturations][1, :], label = "Reference")
lines!(ax, states[end][:Saturations][1, :], label = "Initial guess")
axislegend(ax)
ax = Axis(fig[1, 2], title = "Pressure")
lines!(ax, states_ref[end][:Pressure], label = "Reference")
```

```
lines!(ax, states[end][:Pressure], label = "Initial guess")
axislegend(ax)
fig
```

14.147 Define objective function

Define objective as mismatch between water saturation in current state and reference state. The objective function is currently a sum over all time steps. We implement a function for one term of this sum.

```
step_times = cumsum(case_ref.dt)
function saturation_mismatch(m, state, dt, step_info, forces)
    t = step_info[:time]
    step_no = findmin(x -> abs(x - t), step_times)[2]
    state_ref = states_ref[step_no]
    fld = :Saturations
    val = state[fld]
    ref = state_ref[fld]
    err = 0
    for i in axes(val, 2)
        err += (val[1, i] - ref[1, i])^2
    end
    return dt*err
end
forces = case_ref.forces
dt = case_ref.dt
model = case_ref.model
@assert Jutul.evaluate_objective(saturation_mismatch, model, states_ref, dt, forces) == 0.0
@assert Jutul.evaluate_objective(saturation_mismatch, model, states, dt, forces) > 0.0
```

14.148 Set up a configuration for the optimization

The optimization code enables all parameters for optimization by default, with relative box limits 0.1 and 10 specified here. If use_scaling is enabled the variables in the optimization are scaled so that their actual limits are approximately box limits.

We are not interested in matching gravity effects or viscosity here. Transmissibilities are derived from permeability and varies significantly. We can set log scaling to get a better conditioned optimization system, without changing the limits or the result.

```
cfg = optimization_config(case_dporo, use_scaling = true, rel_min = 0.1, rel_max = 10)
for (ki, vi) in cfg
    if ki in [:TwoPointGravityDifference, :PhaseViscosities, :Transmissibilities]
        vi[:active] = false
    end
end
print_obj = 100;
```

14.149 Set up parameter optimization

This gives us a set of function handles together with initial guess and limits. Generally calling either of the functions will mutate the data Dict. The options are: F_o(x) -> evaluate objective dF_o(dFdx, x) -> evaluate gradient of objective, mutating dFdx (may trigger evaluation of F_o) F_and_dF(F, dFdx, x) -> evaluate F and/or dF. Value of nothing will mean that the corresponding entry is skipped.

```
F_o, dF_o, F_and_dF, x0, lims, data = setup_parameter_optimization(case_dporo, saturation_misma
F_initial = F_o(x0)
dF_initial = dF_o(similar(x0), x0)
@info "Initial objective: $F_initial, gradient norm $(norm(dF_initial))"
```

14.150 Link to an optimizer package

We use Optim.jl but the interface is general enough that e.g. LBFGSB.jl can easily be swapped in.

LBFGS is a good choice for this problem, as Jutul provides sensitivities via adjoints that are inexpensive to compute.

```
import Optim
lower, upper = lims
inner_optimizer = Optim.LBFGS()
opts = Optim.Options(store_trace = true, show_trace = true, time_limit = 30)
results = Optim.optimize(Optim.only_fg!(F_and_dF), lower, upper, x0, Optim.Fminbox(inner_optim
x = results.minimizer
display(results)
F_final = F_o(x)
```

14.151 Compute the solution using the tuned parameters found in x.

```
parameters_t = deepcopy(case_dporo.parameters)
devectorize_variables!(parameters_t, model, x, data[:mapper], config = data[:config])
x_truth = vectorize_variables(case_ref.model, case_ref.parameters, data[:mapper], config = data
states_tuned, = simulate(case_dporo.state0, case_dporo.model, case_dporo.dt, parameters = para
```

14.152 Plot final parameter spread

```
@info "Final residual $F_final (down from $F_initial)"
fig = Figure()
ax1 = Axis(fig[1, 1], title = "Scaled parameters", ylabel = "Value")
scatter!(ax1, x, label = "Final X")
scatter!(ax1, x0, label = "Initial X")
scatter!(ax1, x_truth, label = "Reference X")
lines!(ax1, lower, label = "Lower bound")
```

```
lines!(ax1, upper, label = "Upper bound")
axislegend()
fig
```

14.153 Plot the final solutions.

Note that we only match saturations - so any match in pressure is not guaranteed.

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Saturation")
lines!(ax, states_ref[end][:Saturations][1, :], label = "Reference")
lines!(ax, states[end][:Saturations][1, :], label = "Initial guess")
lines!(ax, states_tuned[end][:Saturations][1, :], label = "Tuned")

axislegend(ax)
ax = Axis(fig[1, 2], title = "Pressure")
lines!(ax, states_ref[end][:Pressure], label = "Reference")
lines!(ax, states[end][:Pressure], label = "Initial guess")
lines!(ax, states_tuned[end][:Pressure], label = "Tuned")
axislegend(ax)
fig
```

14.154 Plot the objective history and function evaluations

```
fig = Figure()
ax1 = Axis(fig[1, 1], yscale = log10, title = "Objective evaluations", xlabel = "Iterations", y
plot!(ax1, data[:obj_hist][2:end])
ax2 = Axis(fig[1, 2], yscale = log10, title = "Outer optimizer", xlabel = "Iterations", ylabel
t = map(x -> x.value, Optim.trace(results))
plot!(ax2, t)
fig
```

14.155 Define an alternative optimization

We can also use the generic interface for optimization, which is a bit less efficient, but a lot more flexible. The previous optimization interface works on numerical parameters (like pore-volumes, transmissibilities, etc.), while the generic interface allows for both numerical and input parameters (like permeability, porosity, etc.).

Note that the viscosity is not defaulted - so we need to let the solver know that it needs to treat it as a distinct parameter, even if we do not optimize on it directly.

```
opt = setup_reservoir_dict_optimization(case_dporo, parameters = [:PhaseViscosities])
free_optimization_parameter!(opt, [:model, :porosity], rel_min = 0.1, rel_max = 10.0)
prm_opt = optimize_reservoir(opt, saturation_mismatch);
opt
```

14.156 Plot the results

```
case_opt = opt.setup_function(prm_opt)
states_opt, = simulate(case_opt);

fig = Figure(size = (1200, 600))
ax = Axis(fig[1, 1], title = "Saturation")
lines!(ax, states_ref[end][:Saturations][1, :], label = "Reference")
lines!(ax, states[end][:Saturations][1, :], label = "Initial guess")
lines!(ax, states_tuned[end][:Saturations][1, :], label = "Tuned (Optim)")
lines!(ax, states_opt[end][:Saturations][1, :], label = "Tuned (reservoir_dict_optimization)")

axislegend(ax)
ax = Axis(fig[1, 2], title = "Pressure")
lines!(ax, states_ref[end][:Pressure], label = "Reference")
lines!(ax, states[end][:Pressure], label = "Initial guess")
lines!(ax, states_tuned[end][:Pressure], label = "Tuned (Optim)")
lines!(ax, states_opt[end][:Pressure], label = "Tuned (reservoir_dict_optimization)")
axislegend(ax)
fig
```

14.157 Plot the resulting porosities

Note that the porosity is only changed in the regions where the front has passed. As the objective function measures the difference in saturation, the objective function is only sensitive to the swept region where the saturation has changed.

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Porosity")
scatter!(ax, prm_opt[:model][:porosity], label = "Optimized porosity")
lines!(ax, states[end][:Saturations][1, :], label = "Final saturation front")
fig
```

14.158 Optimize a single porosity parameter instead

If we introduce the prior assumption that the porosity is constant, we can optimize a single porosity parameter instead of the full porosity field. This is done by writing a setup function that takes a dictionary with the porosity and permeability as input parameters.

```
function setup_bl(d::AbstractDict, step_info = missing)
    return setup_bl(poro = d[:poro], perm = d[:perm])
end

prm_1poro = Dict(:poro => 0.2, :perm => 0.1*si_unit(:darcy))
opt_1poro = setup_reservoir_dict_optimization(prm_1poro, setup_bl)
free_optimization_parameter!(opt_1poro, :poro, rel_min = 0.1, rel_max = 10.0)
prm_opt_1poro = optimize_reservoir(opt_1poro, saturation_mismatch);
```

```
opt_1poro
```

14.159 Plot the results for the single porosity parameter optimization

We observe that we now have recovered the single porosity parameter!

```
fig = Figure()
ax = Axis(fig[1, 1], title = "Porosity")
scatter!(ax, prm_opt[:model][:porosity], label = "Optimized porosity (all cells)")
scatter!(ax, fill(prm_opt_1poro[:poro], 100), label = "Optimized porosity (single parameter)")
lines!(ax, states[end][:Saturations][1, :], label = "Final saturation front")
axislegend()
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Adjoint gradients for the SPE1 model This is a brief example of how to set up a case and compute adjoint gradients for the SPE1 model in a few different ways. These techniques are in general applicable to DATA-type models. ## Load the model

```
using Jutul, JutulDarcy, GeoEnergyIO, GLMakie
data_pth = joinpath(GeoEnergyIO.test_input_file_path("SPE1"), "SPE1.DATA")
data = parse_data_file(data_pth);
case = setup_case_from_data_file(data);
```

14.160 Set up a function to set up the case with custom porosity

We create a setup function that takes in a parameter dictionary `prm` and returns a case with the porosity set to the value in `prm["poro"]`. This is a convenient way to customize a case prior to setup, allowing direct interaction with keywords like PERMX, PORO, etc.

```
function F(prm, step_info = missing)
    data_c = deepcopy(data)
    data_c["GRID"]["PORO"] = fill(prm["poro"], size(data_c["GRID"]["PORO"]))
    case = setup_case_from_data_file(data_c)
    return case
end
```

14.161 Define and simulate truth case (poro from input)

```
x_truth = only(unique(data["GRID"]["PORO"]))
prm_truth = Dict("poro" => x_truth)
case_truth = F(prm_truth)
ws, states = simulate_reservoir(case_truth)
```

14.162 Define pressure difference as objective function

```
function pdiff(p, p0)
    v = 0.0
    for i in eachindex(p)
        v += (p[i] - p0[i])^2
    end
    return v
end

step_times = cumsum(case.dt)
total_time = step_times[end]
function mismatch_objective_p(m, s, dt, step_info, forces)
    t = step_info[:time] + dt
    step = findmin(x -> abs(x - t), step_times)[2]
    p = s[:Reservoir][:Pressure]
    v = pdiff(p, states[step][:Pressure])
    return (dt/total_time)*(v/(si_unit(:bar)*100)^2)
end
```

14.163 Create a perturbed initial guess and optimize

We create a perturbed initial guess for the porosity and optimize the case using the mismatch objective function defined above. The optimization will adjust the porosity to minimize the mismatch between the simulated pressure and the truth case, and recover the original porosity value of 0.3.

```
prm = Dict("poro" => x_truth .+ 0.25)
dprm = setup_reservoir_dict_optimization(prm, F)
free_optimization_parameter!(dprm, "poro", abs_max = 1.0, abs_min = 0.1)
prm_opt = optimize_reservoir(dprm, mismatch_objective_p);
dprm
```

14.164 Plot the optimization history

```
using GLMakie
fig = Figure()
ax = Axis(fig[1, 1], xlabel = "LBFGS iteration", ylabel = "Objective function", yscale = log10)
scatter!(ax, dprm.history.objectives)
fig
```

14.165 Use lumping to match permeability

The model contains three layers with different permeability. We can perturb the permeability in each layer a bit and use the lumping and scaling functionality in the optimizer to recover the original values.

We first define the setup function which expands a permeability value in each cell to PERMX, PERMY and PERMZ, as these are equal for the base model.

```
rmesh = physical_representation(reservoir_domain(case.model))
layerno = map(i -> cell_ijk(rmesh, i)[3], 1:number_of_cells(rmesh))
darcy = si_unit(:darcy)

function F_perm(prm, step_info = missing)
    data_c = deepcopy(data)
    sz = size(data_c["GRID"]["PERMX"])
    permxyz = reshape(prm["perm"], sz)
    data_c["GRID"]["PERMX"] = permxyz
    data_c["GRID"]["PERMY"] = permxyz
    data_c["GRID"]["PERMZ"] = permxyz
    case = setup_case_from_data_file(data_c)
    return case
end
```

14.165.1 Define the starting point

We perturb each layer a bit by multiplying with a constant factor to create a case where the pressure matches.

```
perm_truth = vec(data["GRID"]["PERMX"])
factors = [1.5, 2.0, 5.0]
prm = Dict(
    "perm" => perm_truth.*factors[layerno],
)
```

14.165.2 Optimize with lumping

We can lump parameters of the same name together. In this case, "perm" has one value per cell. We would like to exploit the fact that we know that there should be one unique value per layer as a prior assumption.

We already have a vector with one entry per cell that defines the layers, so we can use this as the lumping parameter directly.

```
perm_opt = setup_reservoir_dict_optimization(prm, F_perm)
free_optimization_parameter!(perm_opt, "perm", rel_min = 0.1, rel_max = 10.0, lumping = layerno
perm_tuned = optimize_reservoir(perm_opt, mismatch_objective_p);
perm_opt
```

14.165.3 Plot the recovered permeability

```
first_entry = map(i -> findfirst(isequal(i), layerno), 1:3)
kval = [perm_truth[first_entry]..., prm["perm"][first_entry]..., perm_tuned["perm"][first_entry
kval = 1000.0.*kval./si_unit(:darcy)
catval = [1, 2, 3, 1, 2, 3, 1, 2, 3]
group = [1, 1, 1, 2, 2, 2, 3, 3, 3]
```

```
colors = Makie.wong_colors()
fig, ax, plt = barplot(catval, kval,
    dodge = group,
    color = colors[group],
    axis = (
        xticks = (1:3, ["Layer 1", "Layer 2", "Layer 3"]),
        ylabel = "Permeability / md",
        title = "Tuned permeability layers"
    ),
)
labels = ["Truth", "Initial guess", "Optimized"]
elements = [PolyElement(polycolor = colors[i]) for i in 1:length(labels)]
title = "Categories"
Legend(fig[1,2], elements, labels, title)
fig
```

14.166 Compute sensitivities outside the optimization

We can also compute the sensitivities outside the optimization process. As our previous setup funciton only has a single parameter (the porosity), we instead switch to the `setup_reservoir_dict_optimization` function that can set up a set of "typical" tunable parameters for any reservoir model. This saves us the hassle of writing this function ourselves when we want to optimize e.g. permeability, porosity and well indices.

```
dprm_case = setup_reservoir_dict_optimization(case)
free_optimization_parameters!(dprm_case)
dprm_grad = parameters_gradient_reservoir(dprm_case, mismatch_objective_p);
```

14.167 Plot the gradient of the mismatch objective with respect to the porosity

We see, as expected, that the gradient is largest in magnitude around the wells and near the front of the displacement.

```
m = physical_representation(reservoir_domain(case.model))
fig, ax, plt = plot_cell_data(m, dprm_grad[:model][:porosity])
ax.title[] = "Gradient of mismatch objective with respect to porosity"
fig
```

14.168 Plot the sensitivities in the interactive viewer

If you are running the example yourself, you can now explore the sensitivities in the interactive viewer. This is useful for understanding how the model responds to changes in the parameters.

```
plot_reservoir(case.model, dprm_grad[:model])
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Geothermal doublet This example demonstrates how to set up a geothermal doublet simulation using JutulDarcy. We will use two different PVT functions–one simple and one realistic–to highlight the importance of accurate fluid physics in geothermal simulations.

```
using Jutul, JutulDarcy, HYPRE, GeoEnergyIO, GLMakie
meter, kilogram, bar, year = si_units(:meter, :kilogram, :bar, :year)
```

14.169 Make setup function

We use the synthethic EGG model egg_model to emulate realistic geology. Instead of using the original wells, we set up a simple injector-producer doublet, placed so that injected fluids will sweep a large part of the reservoir. Set up EGG model

```
egg_dir = JutulDarcy.GeoEnergyIO.test_input_file_path("EGG")
data_pth = joinpath(egg_dir, "EGG.DATA")
case0 = setup_case_from_data_file(data_pth)
domain = reservoir_model(case0.model).data_domain;
```

Make setup function

```
function setup_doublet(sys)

    inj_well = setup_vertical_well(domain, 45, 15, name = :Injector, simple_well = false)
    prod_well = setup_vertical_well(domain, 15, 45, name = :Producer, simple_well = false)

    model = setup_reservoir_model(
        domain, sys,
        thermal = true,
        wells = [inj_well, prod_well],
    );
    rmodel = reservoir_model(model)
    push!(rmodel.output_variables, :PhaseMassDensities, :PhaseViscosities)

    state0 = setup_reservoir_state(model,
        Pressure = 50bar,
        Temperature = convert_to_si(90, :Celsius)
    )

    time = 50year
    pv_tot = sum(pore_volume(reservoir_model(model).data_domain))
    rate = 2*pv_tot/time
    rate_target = TotalRateTarget(rate)
    ctrl_inj  = InjectorControl(rate_target, [1.0],
        density = 1000.0, temperature = convert_to_si(10.0, :Celsius))
```

```
    bhp_target = BottomHolePressureTarget(25bar)
    ctrl_prod = ProducerControl(bhp_target)

    control = Dict(:Injector => ctrl_inj, :Producer => ctrl_prod)

    dt = 4year/12
    dt = fill(dt, Int(time/dt))

    forces = setup_reservoir_forces(model, control = control)

    return JutulCase(model, dt, forces, state0 = state0)

end
```

14.170 Simple fluid physics

We start by setting up a simple fluid physics where water is slightly compressible, but with no influence of temperature. Viscosity is constant.

```
rhoWS = 1000.0kilogram/meter^3
sys = SinglePhaseSystem(AqueousPhase(), reference_density = rhoWS)
case_simple = setup_doublet(sys)
results_simple = simulate_reservoir(case_simple);
```

Interactive plot of the reservoir state

```
plot_reservoir(case_simple.model, results_simple.states)
```

14.171 Realistic fluid physics

Next, we repeat the simulation with more realistic fluid physics. We use a formulation from NIST where density, viscosity and heat capacity depend on pressure and temperature.

```
case_real = setup_doublet(:geothermal)
results_real = simulate_reservoir(case_real);
```

Interactive plot of the reservoir state

```
plot_reservoir(case_real.model, results_real.states)
```

14.172 Compare results

A key performace metric for geothermal doublets is the time it takes before the cold water injected to uphold pressure reaches the producer. At this point, production temperature will rapidly decline, so that the breakthrough time effectively defines the lifespan of the doublet. We plot the well results for the two simulations to compare the two different PVT formulations. Since water viscosty is not affected by temperature in the simple PVT model, water movement is much faster in this scenario, thereby grossly underestimating the lifespan of the doublet compared to the realistic PVT. This

effect is further amplified by the thermal shrinkage due to colling present in the realistic PVT model.

```
plot_well_results([results_simple.wells, results_real.wells]; names = ["Simple", "Realistic"])
```

Finally, we plot the density to see how the two simulations differ. As density in the the simple PVT is only dependent on pressure, it is largely constant except from in the vicinity of the wells, where pressure gradients are larger. In the realistic PVT, where density is a function of both pressure and temperature, we see that it is affected in all regions swept by the injected cold water.

```
 _simple = map(s -> s[:PhaseMassDensities], results_simple.states)
 _real = map(s -> s[:PhaseMassDensities], results_real.states)
Δ  = map(Δ  -> Dict(:DensityDifference => Δ ),  _simple .-  _real)
plot_reservoir(case_real.model, Δ ; step = length(Δ ))
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # High-temperature Aquifer Thermal Energy Storage (HT-ATES) This example demonstrates how to simulate high-temperature aquifer thermal energy storage. We set up a simple case describing a vertical slice of a reservoir with an a main (hot) well near the left boundary, and a supporting (cold) well near the right boundary. The reservoir has boundarys condition that provides a constant pressure and temperature. We set up a yearly cycle where energy is stored from June to September, and discharged from December to March. The rest of the year is a rest period where no energy is stored or produced.

```
using JutulDarcy, Jutul, HYPRE
import Dates: monthname
darcy, litre, year, second = si_units(:darcy, :litre, :year, :second)

nx = 100
nz = 100

temperature_top = convert_to_si(40.0, :Celsius)
pressure_top = convert_to_si(120.0, :bar)
temperature_surface = convert_to_si(10.0, :Celsius)

grad_p = 1000*9.81
grad_T = 0.3
```

14.173 Set up the reservoir

```
g = CartesianMesh((nx, 1, nz), (250.0, 250.0, 75.0))
reservoir = reservoir_domain(g,
    permeability = [0.3, 0.3, 0.1].*darcy,
    porosity = 0.3,
    rock_thermal_conductivity = 2.0,
    fluid_thermal_conductivity = 0.6
```

```
)

depth = reservoir[:cell_centroids][3, :];
```

14.174 Define wells and model

```
di = Int(ceil(nx/4))
k = Int(ceil(nz/2))
Whot = setup_vertical_well(reservoir, 0+di    , 1, toe = k, name = :Hot)
Wcold = setup_vertical_well(reservoir, nx-di+1, 1, toe = k, name = :Cold)

model = setup_reservoir_model(reservoir, :geothermal, wells = [Whot, Wcold]);
```

14.175 Set up boundary and initial conditions

```
bcells = Int[]
pressure_res = Float64[]
temperature_res = Float64[]
for cell in 1:number_of_cells(g)
    d = depth[cell]
    push!(pressure_res, pressure_top + grad_p*d)
    push!(temperature_res, temperature_top + grad_T*d)

    I, J, K = cell_ijk(g, cell)
    if I == 1 || I == nx
        push!(bcells, cell)
    end
end

bc = flow_boundary_condition(bcells, reservoir, pressure_res[bcells], temperature_res[bcells])
```

14.176 Set up the schedule

14.176.1 Set up forces

We assume we have a supply amounting to 90°C. at 25 l/s for storage. During the discharge period, we assume the same discharge rate and a temperature of 10°C.

```
charge_rate = 25litre/second
discharge_rate = charge_rate
temperature_charge = temperature_top + 50.0
temperature_discharge = temperature_top - 30.0
```

Set up forces for charging

```
rate_target = TotalRateTarget(charge_rate)
ctrl_hot  = InjectorControl(rate_target, [1.0], density = 1000.0, temperature = temperature_cha
rate_target = TotalRateTarget(-charge_rate)
ctrl_cold = ProducerControl(rate_target)
forces_charge = setup_reservoir_forces(model, control = Dict(:Hot => ctrl_hot, :Cold => ctrl_co
```

Set up forces for discharging

```
rate_target = TotalRateTarget(discharge_rate)
ctrl_cold = InjectorControl(rate_target, [1.0], density = 1000.0, temperature = temperature_dis
rate_target = TotalRateTarget(-discharge_rate)
ctrl_hot = ProducerControl(rate_target)
forces_discharge = setup_reservoir_forces(model, control = Dict(:Hot => ctrl_hot, :Cold => ctrl
```

14.176.2 Set up forces for rest period

```
forces_rest = setup_reservoir_forces(model, bc = bc)
```

14.176.3 Set up timesteps and assign forces to each timestep

```
num_years = 25
dt = Float64[]
forces = []
month = year/12
for year in 1:num_years
    for mno in vcat(6:12, 1:5)
        mname = monthname(mno)
        if mname in ("January", "February", "March", "December")
            push!(dt, month)
            push!(forces, forces_discharge)
        elseif mname in ("June", "July", "August", "September")
            push!(dt, month)
            push!(forces, forces_charge)
        else
            @assert mname in ("April", "May", "October", "November")
            push!(dt, month)
            push!(forces, forces_rest)
        end
    end
end
```

14.177 Set up initial state

```
state0 = setup_reservoir_state(model, Pressure = pressure_res, Temperature = temperature_res);
```

14.178 Simulate the case

```
ws, states = simulate_reservoir(state0, model, dt,
    forces = forces
);
```

14.179 Plot the reservoir states in the interactive viewer

```
using GLMakie
plot_reservoir(model, states, key = :Temperature, step = num_years*12)
```

14.180 Plot wells interactively

```
plot_well_results(ws)
```

14.181 Plot energy recovery factor

The energy recovery factor is defined as the amount of stored to produced energy. We plot this both cumulatively and for each of the 25 yearly cycles

```
wd = ws.wells[:Hot]
c_p_water = 4.186 # kJ/kgK

well_temp = wd[:temperature]
q = wd[:mass_rate]

storage = q .> 0
q_store = q.*storage
q_prod = q.*(.!storage)
stored_energy = well_temp.*q_store.*c_p_water.*dt
produced_energy = -well_temp.*q_prod.*c_p_water.*dt
 _cumulative = cumsum(produced_energy)./cumsum(stored_energy)
t = cumsum(dt)./si_unit(:day)

 , T = zeros(num_years), zeros(num_years)
for i = 1:num_years
    ix = (1:12) .+ 12*(i-1)
    se = sum(stored_energy[ix])
    pe = sum(produced_energy[ix])
     [i] = pe/se
    T[i] = t[ix[end]]
end

fig = Figure()
ax = Axis(fig[1, 1], title = "Produced energy [kJ]")
```

```
barplot!(ax, T,  , label = "Yearly")
lines!(ax, t,  _cumulative, color = :black, label = "Cumulative")
axislegend(ax, position = :rb)
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Compositional simulation with Clapeyron.jl equations of state This example shows how to set up and run a compositional simulation using equations of state from the Clapeyron.jl package.

The compositional solver in JutulDarcy uses the MultiComponentFlash.jl package for phase equilibrium calculations by defalt, but it is also possible to use other packages that implement the required interface. Here, we demonstrate how to use the JutulDarcy package extension for Clapeyron.jl to set up a compositional simulation using two different equations of state.

```
## Import packages and define helper function
using Jutul, JutulDarcy, GLMakie, MultiComponentFlash

function solve_and_plot_displacement(eos, nx = 50; name = "")
    g = CartesianMesh((nx, 1, 1), (100.0, 10.0, 10.0))
    Darcy, bar, kg, meter, day = si_units(:darcy, :bar, :kilogram, :meter, :day)
    res = reservoir_domain(g, porosity = 0.3, permeability = 0.1*Darcy)
    inj = setup_well(res, 1, name = :Injector)
    prod = setup_well(res, nx, name = :Producer)
    L, V = LiquidPhase(), VaporPhase()
    sys = MultiPhaseCompositionalSystemLV(eos, (L, V))
    model = setup_reservoir_model(res, sys, wells = [inj, prod], extra_out = false);
    state0 = setup_reservoir_state(model, Pressure = 50*bar, OverallMoleFractions = [0.0, 1.0])
    dt = fill(30*day, 12)
    rate_target = TotalRateTarget(sum(pore_volume(res))/ sum(dt))
    I_ctrl = InjectorControl(rate_target, [1.0, 0.0], density = 100.0*kg/meter^3)
    bhp_target = BottomHolePressureTarget(50*bar)
    P_ctrl = ProducerControl(bhp_target)
    controls = Dict()
    controls[:Injector] = I_ctrl
    controls[:Producer] = P_ctrl
    forces = setup_reservoir_forces(model, control = controls)
    _, states = simulate_reservoir(state0, model, dt, forces = forces)

    state = states[end]
    sg = state[:Saturations][2, :]
    z1 = state[:OverallMoleFractions][1, :]
    z2 = state[:OverallMoleFractions][2, :]
    x_c = res[:cell_centroids][1, :]
    fig = Figure()
    ax = Axis(fig[1, 1], xlabel = "x [m]", title = name)
    lines!(ax, x_c, sg, label = "Gas saturation")
```

```
    cnames = MultiComponentFlash.component_names(eos)
    lines!(ax, x_c, z1, label = "C1: $(cnames[1]) mole fraction")
    lines!(ax, x_c, z2, label = "C2: $(cnames[2]) mole fraction")
    axislegend(position = :rc)
    fig
end
```

14.182 Define components and mixture

We here define a binary mixture of methane and carbon dioxide and 50 grid cells for simulation.

```
nx = 50
components = ["carbondioxide", "decane"]
```

14.183 Simulate Peng-Robinson EOS

```
import Clapeyron: PR
solve_and_plot_displacement(PR(components), nx, name = "Peng-Robinson (PR, Clapeyron) EOS")
```

14.184 Simulate Soave-Redlich-Kwong EOS

We can easily switch to another equation of state, e.g., the Soave-Redlich-Kwong EOS. Note that the displacement front is a bit different relative to the Peng-Robinson EOS, which is expected since we have used the EOS without adjusting any parameters for the mixtures.

```
import Clapeyron: SRK
solve_and_plot_displacement(SRK(components), nx, name = "Soave-Redlich-Kwong (SRK, Clapeyron.jl
```

14.185 Conclusion

We see that it is possible to use equations of state from Clapeyron.jl in JutulDarcy to run compositional simulations. The Clapeyron.jl package provides a wide range of equations-of-state that significantly extends the capabilities of the compositional solver.

Note that the default flash used in the compositional solver in JutulDarcy is quite tailored towards coupling with a simulator, both in terms of complexity and code structure. For this reason, other implementations of equations of state may be slower in practice. It is recommended that you test the performance of your chosen equation of state on a test problem like we did here before setting up a large simulation.

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Intro to compositional flow This is a simple conceptual example demonstrating how to solve compositional flow. This example uses a two-component water-CO2 system. Note that the default Peng-Robinson is not accurate for this system without adjustments to the parameters. However, the example

demonstrates the conceptual workflow for getting started with compositional simulation. ## Set up mixture We load the external flash package and define a two-component H2O-CO2 system. The constructor for each species takes in molecular weight, critical pressure, critical temperature, critical volume, acentric factor given as strict SI. This means, for instance, that molar masses are given in kg/mole and not g/mole or kg/kmol.

```
using MultiComponentFlash
h2o = MolecularProperty(0.018015268, 22.064e6, 647.096, 5.595e-05, 0.3442920843)
co2 = MolecularProperty(0.0440098, 7.3773e6, 304.1282, 9.412e-05, 0.22394)

bic = zeros(2, 2)

mixture = MultiComponentMixture([h2o, co2], A_ij = bic, names = ["H2O", "CO2"])
eos = GenericCubicEOS(mixture, PengRobinson())
```

14.186 Set up domain and wells

```
using Jutul, JutulDarcy, GLMakie
nx = 50
ny = 1
nz = 20
dims = (nx, ny, nz)
g = CartesianMesh(dims, (100.0, 10.0, 10.0))
nc = number_of_cells(g)
Darcy, bar, kg, meter, Kelvin, day, sec = si_units(:darcy, :bar, :kilogram, :meter, :Kelvin, :
K = repeat([0.1, 0.1, 0.001]*Darcy, 1, nc)
res = reservoir_domain(g, porosity = 0.3, permeability = K)
```

Set up a vertical well in the first corner, perforated in top layer

```
prod = setup_well(g, K, [(nx, ny, 1)], name = :Producer)
```

Set up an injector in the opposite corner, perforated in bottom layer

```
inj = setup_well(g, K, [(1, 1, nz)], name = :Injector)
```

14.187 Define system and realize on grid

```
rhoLS = 844.23*kg/meter^3
rhoVS = 126.97*kg/meter^3
rhoS = [rhoLS, rhoVS]
L, V = LiquidPhase(), VaporPhase()
sys = MultiPhaseCompositionalSystemLV(eos, (L, V))
model, parameters = setup_reservoir_model(res, sys, wells = [inj, prod], extra_out = true);
push!(model[:Reservoir].output_variables, :Saturations)
kr = BrooksCoreyRelativePermeabilities(sys, 2.0, 0.0, 1.0)
model = replace_variables!(model, RelativePermeabilities = kr)
T0 = fill(303.15*Kelvin, nc)
```

```
parameters[:Reservoir][:Temperature] = T0
state0 = setup_reservoir_state(model, Pressure = 50*bar, OverallMoleFractions = [1.0, 0.0]);
```

14.188 Define schedule

5 year (5*365.24 days) simulation period

```
dt0 = fill(1*day, 26)
dt1 = fill(10.0*day, 180)
dt = cat(dt0, dt1, dims = 1)
rate_target = TotalRateTarget(9.5066e-06*meter^3/sec)
I_ctrl = InjectorControl(rate_target, [0, 1], density = rhoVS)
bhp_target = BottomHolePressureTarget(50*bar)
P_ctrl = ProducerControl(bhp_target)

controls = Dict()
controls[:Injector] = I_ctrl
controls[:Producer] = P_ctrl
forces = setup_reservoir_forces(model, control = controls)
ws, states = simulate_reservoir(state0, model, dt, parameters = parameters, forces = forces);
```

14.189 Once the simulation is done, we can plot the states

Note that this example is intended for static publication in the documentation. For interactive visualization you can use functions like `plot_interactive` to interactively visualize the states.

```
z = states[end][:OverallMoleFractions][2, :]
function plot_vertical(x, t)
    data = reshape(x, (nx, nz))
    data = data[:, end:-1:1]
    fig, ax, plot = heatmap(data)
    ax.title = t
    Colorbar(fig[1, 2], plot)
    fig
end;
```

14.189.1 Plot final CO2 mole fraction

```
plot_vertical(z, "CO2")
```

14.189.2 Plot final vapor saturation

```
sg = states[end][:Saturations][2, :]
plot_vertical(sg, "Vapor saturation")
```

14.189.3 Plot final pressure

```
p = states[end][:Pressure]
plot_vertical(p./bar, "Pressure [bar]")
```

14.189.4 Plot in interactive viewer

```
plot_reservoir(model, states, step = length(dt), key = :Saturations)
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # A more complex compositional model This example sets up a more complex compositional simulation with five different components. Other than that, the example is similar to the others that include wells and is therefore not commented in great detail.

```
using MultiComponentFlash

n2_ch4 = MolecularProperty(0.0161594, 4.58e6, 189.515, 9.9701e-05, 0.00854)
co2 = MolecularProperty(0.04401, 7.3866e6, 304.200, 9.2634e-05, 0.228)
c2_5 = MolecularProperty(0.0455725, 4.0955e6, 387.607, 2.1708e-04, 0.16733)
c6_13 = MolecularProperty(0.117740, 3.345e6, 597.497, 3.8116e-04, 0.38609)
c14_24 = MolecularProperty(0.248827, 1.768e6, 698.515, 7.2141e-04, 0.80784)

bic = [0.11883 0.00070981 0.00077754 0.01 0.011;
       0.00070981 0.15 0.15 0.15 0.15;
       0.00077754 0.15 0 0 0;
       0.01 0.15 0 0 0;
       0.011 0.15 0 0 0]

mixture = MultiComponentMixture([n2_ch4, co2, c2_5, c6_13, c14_24], A_ij = bic, names = ["N2-C
eos = GenericCubicEOS(mixture, PengRobinson())

using Jutul, JutulDarcy, GLMakie
Darcy, bar, kg, meter, Kelvin, day = si_units(:darcy, :bar, :kilogram, :meter, :Kelvin, :day)
nx = ny = 20
nz = 2

dims = (nx, ny, nz)
g = CartesianMesh(dims, (1000.0, 1000.0, 1.0))
nc = number_of_cells(g)
K = repeat([0.05*Darcy], 1, nc)
res = reservoir_domain(g, porosity = 0.25, permeability = K, temperature = 387.45*Kelvin)
```

Set up a vertical well in the first corner, perforated in all layers

```
prod = setup_vertical_well(g, K, nx, ny, name = :Producer)
```

Set up an injector in the opposite corner, perforated in all layers

```
inj = setup_vertical_well(g, K, 1, 1, name = :Injector)

rhoLS = 1000.0*kg/meter^3
rhoVS = 100.0*kg/meter^3

rhoS = [rhoLS, rhoVS]
L, V = LiquidPhase(), VaporPhase()
```

Define system and realize on grid

```
sys = MultiPhaseCompositionalSystemLV(eos, (L, V))
model = setup_reservoir_model(res, sys, wells = [inj, prod], block_backend = true);
kr = BrooksCoreyRelativePermeabilities(sys, 2.0, 0.0, 1.0)
model = replace_variables!(model, RelativePermeabilities = kr)

push!(model[:Reservoir].output_variables, :Saturations)

state0 = setup_reservoir_state(model, Pressure = 225*bar, OverallMoleFractions = [0.463, 0.016
dt = repeat([2.0]*day, 365)
rate_target = TotalRateTarget(0.0015)
I_ctrl = InjectorControl(rate_target, [0, 1, 0, 0, 0], density = rhoVS)
bhp_target = BottomHolePressureTarget(100*bar)
P_ctrl = ProducerControl(bhp_target)

controls = Dict()
controls[:Injector] = I_ctrl
controls[:Producer] = P_ctrl
forces = setup_reservoir_forces(model, control = controls)
ws, states = simulate_reservoir(state0, model, dt, forces = forces);
```

14.190 Once the simulation is done, we can plot the states

14.190.1 CO2 mole fraction

```
sg = states[end][:OverallMoleFractions][2, :]
fig, ax, p = plot_cell_data(g, sg)
fig
```

14.190.2 Gas saturation

```
sg = states[end][:Saturations][2, :]
fig, ax, p = plot_cell_data(g, sg)
fig
```

14.190.3 Pressure

```
p = states[end][:Pressure]
fig, ax, p = plot_cell_data(g, p)
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Consistent discretizations: Average MPFA and nonlinear TPFA This example demonstrates how to use alternative discretizations for the pressure gradient term in the Darcy equation, i.e. the approximation of the Darcy flux:

$K(\nabla p + \rho g \Delta z)$.

It is well-known that for certain combinations of grid geometry and permeability fields, the classical two-point flux approximation scheme can give incorrect results. This is due to the fact that the TPFA scheme is not a formally consistent method when the product of the permeability tensor and the normal vector does not align with the cell-to-cell vectors over a face (lack of K-orthogonality).

In such cases, it is often beneficial to use a consistent discretization. JutulDarcy includes a class of linear and nonlinear schemes that are designed to be accurate even for challenging grids.

For further details on this class of methods, which differ a bit from the classical MPFA-O type method often seen in the literature, see schneider_nonlinear, zhang_nonlinear and raynaud_discretization. ## Define a mesh and twist the nodes This makes the mesh non K-orthogonal and will lead to wrong solutions for the default TPFA scheme.

```
using Jutul
using JutulDarcy
using LinearAlgebra
using GLMakie
using Test # hide

sys = SinglePhaseSystem()
nx = nz = 100
pdims = (1.0, 1.0)
g = CartesianMesh((nx, nz), pdims)

g = UnstructuredMesh(g)
D = dim(g)

v = 0.1
for i in eachindex(g.node_points)
    x, y = g.node_points[i]
    shiftx =  v*sin( *x)*sin(3*(- /2 +  *y))
    shifty =  v*sin( *y)*sin(3*(- /2 +  *x))
    g.node_points[i] += [shiftx, shifty]
end
```

```
nc = number_of_cells(g)
domain = reservoir_domain(g, permeability = 0.1*si_unit(:darcy))

fig = Figure()
Jutul.plot_mesh_edges!(Axis(fig[1, 1]), g)
fig
```

14.191 Create a test problem function

We set up a problem for our given domain with left and right boundary boundary conditions that correspond to a linear pressure drop. We can expect the steady-state pressure solution to be linear between the two faces as there is no variation in permeability or significant compressibility. The function will return the pressure solution at the end of the simulation for a given scheme.

```
function solve_test_problem(scheme)
    model = setup_reservoir_model(domain, sys,
        general_ad = true,
        kgrad = scheme,
        block_backend = false
    )
    state0 = setup_reservoir_state(model, Pressure = 1e5)
    nc = number_of_cells(g)
    bcells = Int64[]
    bpres = Float64[]
    for k in 1:nz
        bnd_l = Jutul.cell_index(g, (1, k))
        bnd_r = Jutul.cell_index(g, (nx, k))
        push!(bcells, bnd_l)
        push!(bpres, 1e5)
        push!(bcells, bnd_r)
        push!(bpres, 2e5)
    end
    bc = flow_boundary_condition(bcells, domain, bpres)
    forces = setup_reservoir_forces(model, bc = bc)

    dt = [si_unit(:day)]
    _, states = simulate_reservoir(state0, model, dt,
        forces = forces, failure_cuts_timestep = false,
        tol_cnv = 1e-6,
        linear_solver = GenericKrylov(preconditioner = AMGPreconditioner(:smoothed_aggregation)
        )
    return states[end][:Pressure]
end
```

14.192 Solve the test problem with three different schemes

- TPFA (two-point flux approximation, inconsistent, linear)

- Average MPFA (consistent, linear)
- NTPFA (nonlinear two-point flux approximation, consistent, nonlinear)

```
results = Dict()
for m in [:tpfa, :avgmpfa, :ntpfa]
    println("Solving $m")
    results[m] = solve_test_problem(m);
end
```

14.193 Plot the results

We plot the pressure solution for each of the schemes, as well as the error. Note that the color axis varies between error plots. As the grid is quite skewed, we observe significant errors for the TPFA scheme, with no significant error for the consistent schemes.

```
x = domain[:cell_centroids][1, :]
get_ref(x) = x*1e5 + 1e5
x_distinct = sort(unique(x))
sol = get_ref.(x_distinct)

fig = Figure(size = (1200, 600))
for (i, (m, s)) in enumerate(results)
    ref = get_ref.(x)
    err = norm(s .- ref)/norm(ref)

    ax = Axis(fig[1, i], title = "$m")
    lines!(ax, x_distinct, sol, color = :red)
    scatter!(ax, x, s, label = m, markersize = 1, alpha = 0.5, transparency = true)

    ax2 = Axis(fig[2, i], title = "Error=$err")
    Δ = s - ref
    emin, emax = extrema(Δ)
    largest = max(abs(emin), abs(emax))
    crange = (-largest, largest)
    plt = plot_cell_data!(ax2, g, Δ, colorrange = crange, colormap = :seismic)
    Colorbar(fig[3, i], plt, vertical = false)
    if m != :tpfa  # hide
        @test err < 1e-4 # hide
    end  # hide
end
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Consistent and high-resolution: WENO, NTPFA and AvgMPFA The following example demonstrates how to set up a reservoir model with different discretizations for flow and transport for a skewed grid. The model setup is taken from 6.1.2 in the MRST book mrst-book-i.

The example is a two-dimensional model with a single layer. The domain itself is completely symmetric, with producer in either lower corner and a single injector in the middle of the domain. From a flow perspective, the model is completely symmetric, and exactly the same results should be obtained for both producer wells. The grid is intentionally distorted to be skewed towards one producer, which will lead to consistency issues for the standard two-point flux approximation used by the solvers.

The fluid model is also quite simple, with two immiscible phases that have linear relative permeability functions. A consequence of this is that the fluid fronts are not self-sharpening and are expected to be smeared out due to the numerical diffusion of the standard first-point upwind scheme.

```
using Jutul, JutulDarcy, GLMakie
Darcy, kg, meter, day, bar = si_units(:darcy, :kg, :meter, :day, :bar)
ny = 20
nx = 2*ny + 1
nz = 1

gcart = CartesianMesh((nx, ny, nz), (2.0, 1.0, 1.0))
g = UnstructuredMesh(gcart)

for (i, pt) in enumerate(g.node_points)
    x, y, z = pt
    pt += [0.4*(1.0-(x-1.0)^2)*(1.0-y), 0, 0]
    g.node_points[i] = pt .* [1000.0, 1000.0, 1.0]
end

c_i = cell_index(g, (nx÷2+1, ny, 1))
c_p1 = cell_index(g, (1, 1, 1))
c_p2 = cell_index(g, (nx, 1, 1))

fig = Figure()
ax = Axis(fig[1, 1])
Jutul.plot_mesh_edges!(ax, g)
plot_mesh!(ax, g, cells = c_i, color = :red)
plot_mesh!(ax, g, cells = [c_p1, c_p2], color = :blue)
fig
```

14.194 Define wells, fluid system and controls

```
reservoir = reservoir_domain(g, permeability = 0.1Darcy, porosity = 0.1)

c_i1 = cell_index(g, (nx÷2+1, ny, 1))

I1 = setup_vertical_well(reservoir, nx÷2+1, ny, name = :I1, simple_well = true)
P1 = setup_vertical_well(reservoir, 1, 1, name = :P1, simple_well = true)
P2 = setup_vertical_well(reservoir, nx, 1, name = :P2, simple_well = true)
```

```
phases = (AqueousPhase(), LiquidPhase())
rhoWS = rhoLS = 1000.0
rhoS = [rhoWS, rhoLS] .* kg/meter^3
sys = ImmiscibleSystem(phases, reference_densities = rhoS)

dt = repeat([30.0]*day, 100)
pv = pore_volume(reservoir)
inj_rate = sum(pv)/sum(dt)

rate_target = TotalRateTarget(inj_rate)
I_ctrl = InjectorControl(rate_target, [1.0, 0.0], density = rhoWS)
bhp_target = BottomHolePressureTarget(50*bar)
P_ctrl = ProducerControl(bhp_target)
controls = Dict()
controls[:I1] = I_ctrl
controls[:P1] = P_ctrl
controls[:P2] = P_ctrl
```

14.195 Define functions to perform the simulations

We are going to perform a number of simulations with different discretizations and it is therefore convenient to setup functions that can be called with the desired discretizations as arguments. The results are stored in a dictionary for easy retrieval after the simulations are done.

```
all_results = Dict()
```

14.195.1 Function to perform simulation

The function `simulate_with_discretizations` sets up the reservoir model with the requested discretizations. The default behavior mirrors the default behavior of JutulDarcy by using the industry standard single-point upwind, two-point flux approximation.

```
function simulate_with_discretizations(upwind = :spu, kgrad = :tpfa)
    model = setup_reservoir_model(reservoir, sys,
        kgrad = kgrad,
        upwind = upwind,
        wells = [P1, P2, I1]
    )
    kr = BrooksCoreyRelativePermeabilities(sys, 1.0, 0.0, 1.0)
    replace_variables!(model, RelativePermeabilities = kr)
    forces = setup_reservoir_forces(model, control = controls)
    state0 = setup_reservoir_state(model, Pressure = 100*bar, Saturations = [0.0, 1.0])
    return simulate_reservoir(state0, model, dt, info_level = 0, forces = forces);
end
```

14.195.2 Function to plot the results

We create a function `plot_discretization_result` that takes the result from a simultation and plots the saturation field after 75 steps.

```
function plot_discretization_result(result, name)
    ws, states = result
    sg = states[75][:Saturations][1, :]
    fig = Figure()
    ax = Axis(fig[1, 1], title = name)
    plot_cell_data!(ax, g, sg, colormap = :seaborn_icefire_gradient)
    Jutul.plot_mesh_edges!(ax, g)
    return fig
end
```

14.195.3 Function to simulate and plot discretizations

The function `simulate_and_plot_discretizations` is a convenience function that calls the `simulate_with_discretizations` function and then plots the result.

```
function simulate_and_plot_discretizations(upwind, kgrad)
    result = simulate_with_discretizations(upwind, kgrad)
    ustr = uppercase("$upwind")
    if kgrad == :avgmpfa
        kgstr = "AverageMPFA"
    else
        kgstr = uppercase("$kgrad")
    end
    descr = "$ustr with $kgstr"
    all_results[descr] = result
    plot_discretization_result(result, descr)
end
```

14.195.4 Simulate SPU-TPFA

We start by simulating the model with the standard single-point upwind and the two-point flux approximation. This is the default discretization used by JutulDarcy. The schemes are robust (easy to implement and obtain nonlinear convergence) but suffer for consistency issues on skewed and non-K-orthogonal grids. We can observe both significant smearing (the fluid front is not sharp) and a significant bias towards the producer in the lower right corner.

```
simulate_and_plot_discretizations(:spu, :tpfa)
```

14.195.5 Simulate SPU-AvgMPFA

The next simulation uses a type of multi-point flux approximation (AverageMPFA or AvgMPFA), which is one class of consistent discretizations. As the scheme is consistent, the effect of the skewed grid is significantly reduced, with the remaining error being due to the coarse mesh chosen for illustrative purposes. Using AverageMPFA does not alleviate the smearing of the fluid front, however.

```
simulate_and_plot_discretizations(:spu, :avgmpfa)
```

14.195.6 Simulate SPU-TPFA

The next simulation uses a nonlinear two-point flux approximation (NTPFA). This scheme is consistent and monotone, and can be derived from the AverageMPFA scheme by allowing the ratio between the two half-face fluxes at each interface to vary based on pressure.

```
simulate_and_plot_discretizations(:spu, :ntpfa)
```

14.195.7 Simulate WENO-TPFA

The penultimate simulation uses a high-resolution scheme (WENO) for the flow, but reverts back to the inconsistent TPFA scheme for the pressure gradient. We now see significantly sharper fronts, but the bias towards the producer in the lower right corner is now back.

```
simulate_and_plot_discretizations(:weno, :tpfa)
```

14.195.8 Simulate WENO-AvgMPFA

The final simulation combines the high-resolution WENO scheme with the consistent AverageMPFA scheme. The result is a sharp fluid front with no bias towards either producer. This is the most accurate simulation of the four for this intentionally badly gridded problem.

```
simulate_and_plot_discretizations(:weno, :avgmpfa)
```

14.196 Compare the results

We can now compare the results of the different discretizations. We start by plotting the well curves in the two producers. Recall that the model is, from a flow perspective, completely symmetric, and the water cut in the two producers should be identical if the problem is fully resolved. We observe this for the consistent schemes, and delayed breakthrough for the SPU solves.

```
fig = Figure()
ax1 = Axis(fig[1, 1], title = "P1 water cut")
ax2 = Axis(fig[2, 1], title = "P2 water cut")
colors = Makie.wong_colors()
for (i, pr) in enumerate(all_results)
    name, result = pr
    ws, states = result
    wcut1 = ws[:P1][:wcut]
    wcut2 = ws[:P2][:wcut]
    lines!(ax1, wcut1, label = "$name", color = colors[i])
    lines!(ax2, wcut2, label = "$name", color = colors[i])
end
axislegend(ax1, position = :lt)
fig
```

14.197 Compare the saturation fields as contours

We can also compare the saturation fields for the different discretizations to see the spatial differences. To make life a bit easier, we make a function to do the comparison plots for us.

```
function compare_contours(name1, name2, title)
    cmp = :seaborn_icefire_gradient
    r1 = all_results[name1]
    r2 = all_results[name2]
    fig = Figure(size = (1200, 800))
    ax = Axis(fig[1, 1], title = title)
    s1 = reshape(r1.states[75][:Saturations][1, :], nx, ny)
    s2 = reshape(r2.states[75][:Saturations][1, :], nx, ny)

    contourf!(ax, s2, colormap = cmp, label = name2)
    contour!(ax, s1, color = :white, linewidth = 5)
    contour!(ax, s1, colormap = cmp, linewidth = 3, label = name1)
    axislegend(position = :ct)

    axd = Axis(fig[1, 2], title = "Difference")
    plt = contourf!(axd, s1-s2,
        colormap = :seismic,
        label = name2,
        levels = range(-1, 1, length = 10),
        colorscale = (-1.0, 1.0)
    )
    Colorbar(fig[1, 3], limits = (-1, 1), colormap = :seismic)
    Colorbar(fig[2, 1:3], limits = (-1, 1), colormap = cmp, vertical = false)

    ax1 = Axis(fig[3, 1], title = name1)
    contourf!(ax1, s1, colormap = cmp)

    ax2 = Axis(fig[3, 2], title = name2)
    contourf!(ax2, s2, colormap = cmp)
    fig
end
```

14.197.1 Compare SPU-TPFA and SPU-AvgMPFA

```
compare_contours("SPU with AverageMPFA", "SPU with TPFA", "Consistent (AvgMPFA) vs inconsisten
```

14.197.2 Compare WENO-TPFA and SPU-TPFA

```
compare_contours("WENO with TPFA", "SPU with TPFA", "High resolution (WENO) vs first order (SPU
```

14.197.3 Compare WENO-AvgMPFA and SPU-TPFA

```
compare_contours("WENO with AverageMPFA", "SPU with TPFA", "High resolution + consistent vs fi
```

14.198 Conclusion

The example demonstrates how different discretizations can affect the results of a simulation. The example is intentionally set up to show the differences in the discretizations. The results show that the consistent discretizations (AvgMPFA and NTPFA) can reduce the bias from grid orientation, and, while not present here, permeability tensor effects. The high-resolution WENO scheme can be employed to reduce smearing of the fluid fronts, which is important for problems that are not self-sharpening (e.g. compositional flow and miscible displacements).

These schemes are not a pancea, however, as they require more computational effort to linearize and can have robustness issues where nonlinear convergence rates deteriorate and shorter timesteps may be required. They are accessible in JutulDarcy through the high-level interface and can be applied to any mesh and physics combination supported by the rest of the solvers.

We plot a summary of the saturation fields for all combinations of a pair of consistent and a pair of inconsistent discretizations to demonstrate the key differences.

```
fig = Figure(size = (1200, 800))
function plot_sat!(i, j, name)
    sg = all_results[name].states[75][:Saturations][1, :]
    ax = Axis(fig[i,j])
    hidespines!(ax)
    hidedecorations!(ax)
    plt = plot_cell_data!(ax, g, sg, colormap = :seaborn_icefire_gradient, colorrange = (0.0, 
    Jutul.plot_mesh_edges!(ax, g)
    return plt
end
Label(fig[0, 1], "Single-point upwind", fontsize = 30, tellheight = true, tellwidth = false)
Label(fig[0, 2], "High resolution", fontsize = 30, tellheight = true, tellwidth = false)

Label(fig[1, 0], "TPFA", fontsize = 30, rotation = pi/2, tellheight = false, tellwidth = true)
Label(fig[2, 0], "AverageMPFA", fontsize = 30, rotation = pi/2, tellheight = false, tellwidth =

plot_sat!(1, 1, "SPU with TPFA")
plot_sat!(2, 1, "SPU with AverageMPFA")
plot_sat!(1, 2, "WENO with TPFA")
plt = plot_sat!(2, 2, "WENO with AverageMPFA")
Colorbar(fig[:, 3], plt)
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # CO2-brine correlations with salinity JutulDarcy includes a set of pre-generated tables for simulation of CO2

storage in saline aquifers, as well as functions for calculating PVT and solubility properties of CO2-brine mixtures with varying degrees of salinity.

For more details on the property calculations used herein, please see the paper Three-dimensional simulation of geologic carbon dioxide sequestration using MRST by L. Saló et al (2024).

This example demonstrates how to plot the properties, and does a basic comparison of how the properties change when the aqueous phase has salts added.

```
using Jutul, JutulDarcy, GLMakie
import JutulDarcy.CO2Properties: co2_brine_property_tables
```

14.199 Define plotting functions

We define a plotting function that varies temperature, and one that compares the same property with and without salts for a given temperature.

```
T = [20.0, 40.0, 60.0, 80.0, 100.0]
p = range(5, 300, 100)

function plot_property(prop, tab, component, title = "")
    trans = x -> x
    if prop == :viscosity
        u = "(Pa s)"
    elseif prop == :density
        u = "(kg/m^3)"
    elseif prop == :K
        u = " (log10)"
        trans = log10
    else
        u = ""
    end
    is_h2o = component == :H2O
    fig = Figure(size = (800, 600))
    ax = Axis(fig[1, 1], title = title, xlabel = "Pressure (bar)", ylabel = "$component $prop $

    F = tab[prop]
    if prop == :K
        F = F.K
    end
    for (i, T_i) in enumerate(T)
        val = F.(convert_to_si(p, :bar), T_i + 273.15)
        if is_h2o
            val = map(first, val)
        else
            val = map(last, val)
        end
        lines!(ax, p, trans.(val), label = "T=$(T_i) °C",
            colormap = :hot,
```

```
            color = i,
            colorrange = (1, length(T)+3),
            linewidth = 2,
        )
    end
    axislegend(position = :lt, orientation = :horizontal)
    fig
end

function plot_brine_comparison(prop, T, tab1, tab2, component, title = "")
    trans = x -> x
    if prop == :viscosity
        u = "(Pa s)"
    elseif prop == :density
        u = "(kg/m^3)"
    elseif prop == :K
        u = ""
        u = " (log10)"
        trans = log10
    else
        u = ""
    end
    is_h2o = component == :H2O
    fig = Figure(size = (800, 600))
    ax = Axis(fig[1, 1], title = title, xlabel = "Pressure (bar)", ylabel = "$component $prop $

    pbar = convert_to_si.(p, :bar)

    F1 = tab1[prop]
    F2 = tab2[prop]
    if prop == :K
        F1 = F1.K
        F2 = F2.K
    end
    val1 = F1.(pbar, T + 273.15)
    val2 = F2.(pbar, T + 273.15)

    if is_h2o
        val1 = map(first, val1)
        val2 = map(first, val2)
    else
        val1 = map(last, val1)
        val2 = map(last, val2)
    end
    lines!(ax, trans.(val1), label = "Water",
        linewidth = 2,
    )
```

```
    lines!(ax, trans.(val2), label = "Brine",
        linewidth = 2,
    )
    axislegend(position = :lt, orientation = :horizontal)
    fig
end
```

14.200 Generate tables

We get tables for a wide pressure and temperature range. The first set of tables assumes no salts, and the second set of tables uses specified molar fractions of salts in the aqueous phase.

```
tab_water = co2_brine_property_tables()
tab_brine = co2_brine_property_tables(
    salt_names = ["NaCl", "KCl"],
    salt_mole_fractions = [0.05, 0.01]
)
```

14.201 Plot aqueous mass density

```
plot_property(:density, tab_water, :H2O, "Pure phase density, no salts")
```

14.202 Plot gaseous mass density

```
plot_property(:density, tab_water, :CO2, "Pure phase density, no salts")
```

14.203 Plot aqueous viscosity

```
plot_property(:viscosity, tab_water, :H2O, "Pure phase viscosity, no salts")
```

14.204 Plot gaseous viscosity

```
plot_property(:viscosity, tab_water, :CO2, "Pure phase viscosity, no salts")
```

14.205 Plot K-value of CO2

The K value defines the ratio between liquid and vapor phases at equilibrium conditions and is closely related to solubility. For a component we relate the liquid mass fraction x_i to the vapor mole fraction y_i of that component by the K-value K_i:

$y_i = K_i(p, T)x_i$

```
plot_property(:K, tab_water, :CO2, "K value of CO2 component in aqueous phase, no salts")
```

14.206 Compare K value with and without salts

```
plot_brine_comparison(:K, 30.0, tab_water, tab_brine, :CO2, "K value of CO2 component in aqueou
```

14.207 Compare density with and without salts

```
plot_brine_comparison(:density, 30.0, tab_water, tab_brine, :H2O, "Pure phase density")
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Relative Permeabilities in JutulDarcy We will show a few of the functions for evaluating relative permeabilities, show how these can be fitted to data and finally how they can be used in simulation.

```
using GLMakie
using JutulDarcy, Jutul
import JutulDarcy: table_to_relperm, PhaseRelativePermeability, brooks_corey_relperm, let_relpe
```

14.208 Define helper functions

The following functions are used to set up a simple reservoir model with a linear displacement for a given relative permeability function. The function also takes in the viscosity ratio as an optional parameter.

14.208.1 Simulation helpers

```
function setup_model_with_relperm(kr; mu_ratio = 1.0)
    mesh = CartesianMesh(1000, 1000.0)
    domain = reservoir_domain(mesh)
    nc = number_of_cells(domain)
    sys = ImmiscibleSystem((LiquidPhase(), VaporPhase()), reference_densities = [1000.0, 700.0]
    model = setup_reservoir_model(domain, sys)
    rmodel = reservoir_model(model)
    replace_variables!(rmodel, RelativePermeabilities = kr)
    JutulDarcy.add_relperm_parameters!(rmodel)
    parameters = setup_parameters(model)
    parameters[:Reservoir][:PhaseViscosities] = repeat([mu_ratio*1e-3, 1e-3], 1, nc)
    return (model, parameters)
end

function simulate_bl(model, parameters; pvi = 0.75)
    domain = reservoir_domain(model)
```

```
    nc = number_of_cells(domain)
    nstep = nc
    timesteps = fill(3600*24/nstep, nstep)
    p0 = 100*si_unit(:bar)
    tot_time = sum(timesteps)
    pv = pore_volume(domain)
    irate = pvi*sum(pv)/tot_time
    src  = SourceTerm(1, irate, fractional_flow = [1.0, 0.0])
    bc = FlowBoundaryCondition(nc, p0/2)
    forces = setup_reservoir_forces(model, sources = src, bc = bc)
    state0 = setup_reservoir_state(model, Pressure = p0, Saturations = [0.0, 1.0])
    ws, states = simulate_reservoir(state0, model, timesteps,
        forces = forces, parameters = parameters, info_level = -1)
    return states[end][:Saturations][1, :]
end

function define_relperm_single_region(krw_t, krow_t, sw_t)
    krow = PhaseRelativePermeability(reverse(1.0 .- sw), reverse(krow_t), label = :ow)
    krw = PhaseRelativePermeability(sw, krw_t, label = :ow)
    return ReservoirRelativePermeabilities(w = krw, ow = krow)
end
```

14.208.2 Single region table plotting functions

```
function simulate_and_plot(sw, krw, krow, name)
    fig = Figure(size = (1200, 800))
    ax = Axis(fig[1, 1], title = name)
    colors = Makie.wong_colors()
    lines!(ax, sw, krw, label = L"K_{rw}", color = colors[1], linewidth = 2)
    lines!(ax, sw, krow, label = L"K_{row}", color = colors[1], linestyle = :dash, linewidth =

    axislegend(position = :ct)

    krdef = define_relperm_single_region(krw, krow, sw)
    ax = Axis(fig[2, 1], title = L"\text{Injection front at 0.75 PVI}")

    for mu_ratio in [10, 5, 1, 0.2, 0.1]
        println("Simulating with mu_ratio = $mu_ratio")
        model, parameters = setup_model_with_relperm(krdef, mu_ratio = mu_ratio)
        water_front = simulate_bl(model, parameters)
        lines!(ax, water_front, label = L"\mu_w/\mu_o=%$mu_ratio")
    end
    axislegend()
    return fig
end
```

14.209 Brooks-Corey and LET relative permeabilities

We will now show how to use the `brooks_corey_relperm` and `let_relperm` to generate relative permeability tables. These tables can in principle be used in any simulator that supports relative permeability tables.

We define a saturation range and calculate the relative permeabilities for two phases. We then simulate a simple 1D displacement and plot the water front.

14.209.1 Brooks-Corey

The Brooks-Corey model is a simple model that can be used to generate relative permeabilities. The model is defined in the mobile region as:

$k_{rw} = k_{max,w}\bar{S}_w$

$k_{ro} = k_{max,o}\bar{S}_o$

where $k_{max,w}$ is the maximum relative permeability, $\bar{S}_w$ is the normalized saturation for the water phase,

$\bar{S}_w = \frac{S_w - S_{wi}}{1 - S_{wi} - S_{ro}}$

and, similarly, for the oil phase:

$\bar{S}_o = \frac{S_o - S_{ro}}{1 - S_{wi} - S_{ro}}$

JutulDarcy contains a function `brooks_corey_relperm` that can be used to evaluate the values for a given saturation range:

```
sw = range(0, 1, 100)
so = 1.0 .- sw

swi = 0.1
srow = 0.15
r_tot = swi + srow

krw = brooks_corey_relperm.(sw, n = 2, residual = swi, residual_total = r_tot)
krow = brooks_corey_relperm.(so, n = 2, residual = srow, residual_total = r_tot)

simulate_and_plot(sw, krw, krow, L"\text{Brooks-Corey} (N_w = N_{ow} = 2)")
```

14.209.2 Brooks-Corey: Different exponents

We can also use different exponents for the water and oil phases. Here we pick other values for the exponents:

```
krw = brooks_corey_relperm.(sw, n = 4, residual = swi, residual_total = r_tot)
krow = brooks_corey_relperm.(so, n = 1.5, residual = srow, residual_total = r_tot)

simulate_and_plot(sw, krw, krow, L"\text{Brooks-Corey} (N_w = 4, N_{ow} = 1.5)")
```

14.209.3 LET relative permeabilities

The LET model is a generalization of the Brooks-Corey model that introduces additional parameters to more easily control the shape of the curve. It is defined as:

$k_{rw} = k_{max,w} \frac{\bar{S}_w^{L_w}}{\bar{S}_w^{L_w} + E_w(1-\bar{S}_w)^{T_w}}$

The oil phase is defined analogously. The LET model has three exponents L, E, and T that together define the shape of the curve. ### LET table as fully linear We can use the `let_relperm` function to generate a fully linear relative permeability curve, i.e. $k_{rw} = S_w$ and $k_{ro} = S_o$. This is overkill for the LET model, but it is a good way to verify that the function works as expected and that the linear model can be captured by LET functions.

```
krw = let_relperm.(sw, L = 1.0, E = 1.0, T = 1.0, residual = swi, residual_total = r_tot)
krow = let_relperm.(so, L = 1.0, E = 1.0, T = 1.0, residual = srow, residual_total = r_tot)

simulate_and_plot(sw, krw, krow, L"\text{LET relperm (linear)}")
```

14.209.4 LET table, nonlinear exponents

The functions are immediately more interesting if we use nonlinear exponents.

```
krw = let_relperm.(sw, L = 3.0, E = 2.0, T = 1.0, residual = swi, residual_total = r_tot)
krow = let_relperm.(so, L = 2.0, E = 1.0, T = 2.0, residual = srow, residual_total = r_tot)

simulate_and_plot(sw, krw, krow, L"\text{LET relperm (nonlinear)}")
```

14.210 Fitting relative permeabilities to data

We can use either the Brooks-Corey or LET models to fit relative permeabilities to data. We generate some synthetic data and define a function that gives the sum of squares mismatch function between the data and the chosen LET model. As these functions are relatively simple Julia functions, we can use the `Optim` package to perform optimization to find the best fit.

We do this by setting up an objective function that takes the current LET parameters as a flat vector with length five and returns the sum of squares. As Julia is a pretty fast programming language, performing a thousand gradient-free evaluations should only take a few milliseconds, but we could also have used a differentiable optimizer if we wanted to.

The optimizer is provided lower limits for the parameters to avoid numerical issues. The initial guess is a completely linear LET function.

```
import Optim

s_to_match  = [0.1, 0.15, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
kr_to_match = [0.0, 0.0, 0.015, 0.025, 0.1, 0.2, 0.4, 0.6, 0.90, 0.90, 0.90]

function relperm_mismatch(x)
    L, E, T, r, kr_max = x
    kr = let_relperm.(s_to_match, L = L, E = E, T = T, residual = r, kr_max = kr_max)
    return sum((kr .- kr_to_match).^2)
```

```
end

x0 = [1.0, 1.0, 1.0, 0.0, 1.0]
lower = [0.0, 0.0, 0.0, 0.0, 0.1]
upper = [20, 20, 20, 0.5, 1.0]
res = Optim.optimize(relperm_mismatch, lower, upper, x0, Optim.NelderMead(), Optim.Options(ite
L, E, T, r, kr_max = Optim.minimizer(res)
res
```

14.210.1 Plot the matched function

We get a pretty good match - the optimization is able to find reasonable parameters without sign of overfitting. This is because the LET functions are designed to always give a physically reasonable relative permeability curve.

```
fmt = x -> round(x, digits = 2)
println("Optimized parameters:\n\tL = $(fmt(L))\n\tE = $(fmt(E))\n\tT = $(fmt(T))\n\tr = $(fmt
fig = Figure(size = (800, 600))
ax = Axis(fig[1, 1], title = "Optimized LET Relative Permeability")

s_fine = range(0, 1, 1000)
initial_kr = let_relperm.(s_fine, L = x0[1], E = x0[2], T = x0[3], residual = x0[4], kr_max = 
optimized_kr = let_relperm.(s_fine, L = L, E = E, T = T, residual = r, kr_max = kr_max)
scatter!(ax, s_to_match, kr_to_match, label = "Data")
lines!(ax, s_fine, optimized_kr, label = "Optimized LET")
lines!(ax, s_fine, initial_kr, label = "Initial LET")
axislegend(position = :ct)
fig
```

14.211 Multiple regions in JutulDarcy

Relative permeabilities are typically associated with the rock type of the reservoir. If multiple rock types are present, they should be given different relative permeability functions. This can be done by defining a region vector that indices the rock type of each cell. We also define the relative permeability of each phase to be a `Tuple` of two relative permeability functions, one for each region.

We set up utility functions for setting up a two-region relative permeability and then demonstrate this with a simple example that uses Brooks-Corey.

```
function define_relperm_two_regions(krw_t1, krow_t1, krw_t2, krow_t2, sw_t, reg)
    krow1 = PhaseRelativePermeability(reverse(1.0 .- sw), reverse(krow_t1), label = :ow)
    krw1 = PhaseRelativePermeability(sw, krw_t1, label = :ow)
    krow2 = PhaseRelativePermeability(reverse(1.0 .- sw), reverse(krow_t2), label = :ow)
    krw2 = PhaseRelativePermeability(sw, krw_t2, label = :ow)

    return ReservoirRelativePermeabilities(w = (krw1, krw2), ow = (krow1, krow2), regions = reg
end
```

```
function simulate_and_plot_two_regions(sw, krw1, krow1, krw2, krow2, name)
    fig = Figure(size = (1200, 800))
    ax = Axis(fig[1, 1], title = name)
    colors = Makie.wong_colors()
    lines!(ax, sw, krw1, label = L"K_{rw}^1", color = colors[1], linewidth = 2)
    lines!(ax, sw, krow1, label = L"K_{row}^1", color = colors[1], linestyle = :dash, linewidth
    lines!(ax, sw, krw2, label = L"K_{rw}^2", color = colors[2], linewidth = 2)
    lines!(ax, sw, krow2, label = L"K_{row}^2", color = colors[2], linestyle = :dash, linewidth
    axislegend(position = :ct)
    reg = ones(Int, 1000)
    reg[250:end] .= 2
    krdef = define_relperm_two_regions(krw1, krow1, krw2, krow2, sw, reg)
    ax = Axis(fig[2, 1], title = L"\text{Injection front at 0.75 PVI}")
    for mu_ratio in [10, 5, 1, 0.2, 0.1]
        println("Simulating with mu_ratio = $mu_ratio")
        model, parameters = setup_model_with_relperm(krdef, mu_ratio = mu_ratio)
        water_front = simulate_bl(model, parameters)
        lines!(ax, water_front, label = L"\mu_w/\mu_o=%$mu_ratio")
    end
    vspan!(ax, 0, 250, color = (:red, 0.1), label = "Region 1")
    vspan!(ax, 250, 1000, color = (:green, 0.1), label = "Region 2")
    axislegend()
    fig
end

swi = 0.1
srow = 0.15
r_tot = swi + srow

krw1 = brooks_corey_relperm.(sw, n = 2, residual = swi, residual_total = r_tot)
krow1 = brooks_corey_relperm.(so, n = 3, residual = srow, residual_total = r_tot)

swi2 = 0.05
srow2 = 0.2
r_tot2 = swi2 + srow2

krw2 = brooks_corey_relperm.(sw, n = 1.5, residual = swi2, residual_total = r_tot2)
krow2 = brooks_corey_relperm.(so, n = 2.2, residual = srow2, residual_total = r_tot2)

simulate_and_plot_two_regions(sw, krw1, krow1, krw2, krow2, L"\text{Two regions}")
```

14.212 SWOF/SGOF-type tables

The `table_to_relperm` function can be used to convert SWOF/SGOF tables to the JutulDarcy format. The function takes in a table and a saturation range and returns a tuple of two relative permeability functions.

We demonstrate this by using a simple SWOF table and then feed this to the converter before simulating a bit.

Note that JutulDarcy automatically sets up these functions when reading DATA files, so this is only necessary if you want to use the functions in a custom workflow.

```
swof = hcat(
    range(0, 1, length=10),
    [0.0, 0.1, 0.15, 0.25, 0.35, 0.45, 0.55, 0.7, 0.85, 1.0],
    reverse(range(0, 1, length=10)),
    zeros(10)
)
```

14.212.1 Convert to relperm objects and show

```
krw, krow = table_to_relperm(swof)
krw
```

14.212.2 Feed SWOF table to simulation

```
krdef = ReservoirRelativePermeabilities(w = krw, ow = krow)

model, prm = setup_model_with_relperm(krdef)
lines(simulate_bl(model, prm), axis = (title = "SWOF simulation",))
```

14.213 Parametric relative permeabilities

So far, we have been using tabulated relative permeabilities. JutulDarcy is not limited to only working with tables and we can demonstrate this by making use of parametric relative permeabilities instead.

Parametric functions expose parameters such as L, E, T, and residual saturations on a cell-by-cell basis and evaluates the relative permeabilities during simulation with the same functions we used to generate tables in the preceeding section.

Using parametric functions has a few advantages: 1. The relative permeability functions are analytical, which can be more favorable for numerical convergence when compared to standard tables. 2. Exposing the parameters through Jutul's parameter system makes it possible to perform gradient-based history matching. 3. The parameters can vary from cell to cell, which can be useful for reduced-order modelling where the link between relative permeabilities and rock types is disregarded.

```
kr_let = JutulDarcy.ParametricLETRelativePermeabilities()
model, prm = setup_model_with_relperm(kr_let)
```

```
lines(simulate_bl(model, prm), axis = (title = "Parametric LET function simulation",))
```

14.213.1 Check out the parameters

The LET parameters are now numerical parameters in the reservoir:

```
rmodel = reservoir_model(model)
```

14.214 Conclusion

We have explored a few aspects of relative permeabilities in JutulDarcy. There are a number of advanced topics that we did not cover, such as the use of end-point scaling in the more conventional reservoir simulation sense or the use of hysteresis. We still hope that this example has given you a good starting point for working with relative permeabilities in JutulDarcy.

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Overview

JutulDarcy.jl uses industry standard discretizations and will match other simulators when the parameters and driving forces are identical. Setting up identical parameters can be challenging, however, as there are many subtleties in exactly how input is interpreted. In this section, we go over some things to keep in mind when doing a direct comparison to other simulators.

14.215 Comparison of simulations

If you want to compare two numerical models it is advisable to start with a simple physical process that you fully understand how to set up in both simulators. In this section, we provide some hints to help this process.

14.215.1 Input files

The easiest way to perform a one-to-one comparison is through input files (.DATA files). JutulDarcy takes care to interpret these files in a similar manner to commercial codes, including features like transmissibility multipliers, block-to-block transmissibilities and processing of corner-point meshes around faults and eroded layers. Take note of yellow text during model setup - this can indicate unsupported or partially supported features. Some of these messages may refer to features that have little or no impact on simulation outcome, and others may have substantial impacts. For a direct comparison we recommend removing the keywords that give warnings and run the modified files in both JutulDarcy and the comparison simulator. Note that different simulators can sometimes also handle defaulted keywords differently, even when both simulators come from the same vendor. If you are experiencing mismatch, make sure that the case is not dependent on any defaulted values.

JutulDarcy does in general not make use of keywords from input files that describe the solution strategy like linear or nonlinear tolerance adjustments, tuning of time-stepping, switching to IMPES/AIM, etc.

14.215.2 Time-steps

JutulDarcy performs dynamic timestepping, taking care to hit the provided report steps to provide output. For large report steps, different simulators can take different time-steps which results in differences in results. Some simulators, like Eclipse, also report sparse data like well results at a higher interval than the requested reporting interval which can make direct comparison of plots tricky. To get higher resolution in the same manner for JutulDarcy, setting `output_substates = true` is useful.

14.215.3 Tolerances

The default tolerances in JutulDarcy are fairly strict, with measures for both point-wise and integral errors. Direct comparison may require checking that convergence criteria are set to similar values.

14.215.4 Parallelism

If runtimes are to be compared, make sure that the simulators are using a similar model for parallelization. The fastest solves are generally achieved with either MPI or GPU parallelization, both of which require a bit of extra effort to set up.

14.216 Downloadable examples

The remainder of this part of the manual are a set of examples where JutulDarcy is compared against other simulators that are widely in use (e.g. OPM Flow, MRST, AD-GPRS, Eclipse 100/300). Extensive validation has also been performed on non-public models that for obvious reason cannot be directly incorporated in the documentation. All these examples automatically download the prerequisite data together with one or more results to compare with. The examples should be directly reproducible, but may require additional installation/licenses to produce the comparison results.

14.216.1 User examples

If you have an example that can be shared, either for the documentation or for regression testing, please open an issue. If you have a confidential model with a known solution that can be shared privately, feel free to reach out directly. # Validation of equation-of-state compositional flow This example solves a 1D two-phase, three component miscible displacement problem and compares against existing simulators (E300, AD-GPRS) to verify correctness.

The case is loaded from an input file that can be run in other simulators. For convenience, we provide solutions from the other simulators as a binary file to perform a comparison without having to run and convert results from other the simulators.

This case is a small compositional problem inspired by the examples in Voskov et al (JPSE, 2012). A 1D reservoir of 1,000 meters length is discretized into 1,000 cells. The model initially contains a mixture made up of 0.6 parts C10, 0.1 parts CO2, and 0.3 parts C1 by moles at 150 degrees C and 75 bar pressure. Wells are placed in the leftmost and rightmost cells of the domain, with the leftmost well injecting pure CO_2 at a fixed bottom-hole pressure of 100 bar and the other well producing at 50 bar. The model is isothermal and contains a phase transition from the initial two-phase mixture to single-phase gas as injected CO_2 eventually displaces the resident fluids. For further details on this setup, see Møyner and Tchelepi (SPE J. 2018) moyner_tchelepi_2018.

```
using JutulDarcy
using Jutul
using GLMakie
dpth = JutulDarcy.GeoEnergyIO.test_input_file_path("SIMPLE_COMP")
data_path = joinpath(dpth, "SIMPLE_COMP.DATA")
case = setup_case_from_data_file(data_path)
result = simulate_reservoir(case)
ws, states = result;
```

14.217 Plot solutions and compare

The 1D displacement can be plotted as a line plot. We pick a step midway through the simulation and plot compositions, saturations and pressure.

```
cmap = :tableau_hue_circle
ref_path = joinpath(dpth, "reference.jld2")
ref = Jutul.JLD2.load(ref_path)

step_to_plot = 250
fig = with_theme(theme_latexfonts()) do
    x = reservoir_domain(case)[:cell_centroids][1, :]
    mz = 3
    ix = step_to_plot
    mt = :circle
    fig = Figure(size = (800, 400))
    ax = Axis(fig[2, 1], xlabel = "Cell center / m")
    cnames = ["DECANE", "CO2", "METHANE"]
    cnames = ["C ", "CO ", "C "]
    cnames = [L"\text{C}_{10}", L"\text{CO}_2", L"\text{C}_1"]
    lineh = []
    lnames = []
    crange = (1, 4)
    for i in range(crange...)
        if i == 4
            cname = L"\text{S}_g"
            gprs = missing
            ecl = ref["e300"][ix][:Saturations][2, :]
            ju = states[ix][:Saturations][2, :]
        else
            @assert i <= 4
            ecl = ref["e300"][ix][:OverallMoleFractions][i, :]
            gprs = ref["adgprs"][ix][:OverallMoleFractions][i, :]
            ju = states[ix][:OverallMoleFractions][i, :]
            cname = cnames[i]
        end
        h = lines!(ax, x, ju, colormap = cmap, color = i, colorrange = crange, label = cname)
        push!(lnames, cname)
```

```
        push!(lineh, h)
        scatter!(ax, x, ecl, markersize = mz, colormap = cmap, color = i, colorrange = crange)
        if !ismissing(gprs)
            lines!(ax, x, gprs, colormap = cmap, color = i, colorrange = crange, linestyle = :
        end
    end

    l_ju = LineElement(color = :black, linestyle = nothing)
    l_ecl = MarkerElement(color = :black, markersize = mz, marker = mt)
    l_gprs = LineElement(color = :black, linestyle = :dash)

    Legend(
        fig[1, 1],
        [[l_ju, l_ecl, l_gprs], lineh],
        [[L"\text{JutulDarcy}", L"\text{E300}", L"\text{AD-GPRS}"], lnames],
        ["Simulator", "Result"],
        tellwidth = false,
        orientation = :horizontal,
    )
    fig
end
```

14.218 Calculate sensitivities

We demonstrate how the parameter sensitivities of an objective function can be calculated for a compositional model.

The objective function is taken to be the average gas saturation at a specific report step that was plotted in the previous section.

```
import Statistics: mean
import JutulDarcy: reservoir_sensitivities
function objective_function(model, state, Δt, step_info, forces)
    if step_info[:step] != step_to_plot
        return 0.0
    end
    sg = @view state.Reservoir.Saturations[2, :]
    return mean(sg)
end
data_domain_with_gradients = reservoir_sensitivities(case, result, objective_function, include

fig = with_theme(theme_latexfonts()) do
    x = reservoir_domain(case)[:cell_centroids][1, :]
    mz = 3
    ix = step_to_plot
    cmap = :Dark2_5
    cmap = :Accent_4
    cmap = :Spectral_4
```

```
    cmap = :tableau_hue_circle
    mt = :circle
    fig = Figure(size = (800, 400))
    normalize = x -> x./(maximum(x) - minimum(x))
    logscale = x -> sign.(x).*log10.(abs.(x))

    T = data_domain_with_gradients[:temperature]
      = data_domain_with_gradients[:porosity]

    ax1 = Axis(fig[2, 1], yticklabelcolor = :blue, xlabel = "Cell center / m")
    ax2 = Axis(fig[2, 1], yticklabelcolor = :red, yaxisposition = :right)
    hidespines!(ax2)
    hidexdecorations!(ax2)

    l1 = lines!(ax1, x,  T, label = "Temperature", color = :blue)
    l2 = lines!(ax2, x,  , label = "Porosity", color = :red)

    Legend(fig[1, 1], [l1, l2], ["Temperature", "Porosity"], "Parameter", tellwidth = false, o
    fig
end
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # The Egg model: Two-phase oil-water model A two-phase model that is taken from the first member of the EGG ensemble. The model is a synthetic case with channelized permeability and water injection with fixed controls. For more details, see the paper where the ensemble is introduced:

Jansen, Jan-Dirk, et al. "The egg model–a geological ensemble for reservoir simulation." Geoscience Data Journal 1.2 (2014): 192-195.

```
using Jutul, JutulDarcy, GLMakie, DelimitedFiles, HYPRE
egg_dir = JutulDarcy.GeoEnergyIO.test_input_file_path("EGG")
case = setup_case_from_data_file(joinpath(egg_dir, "EGG.DATA"))
ws, states = simulate_reservoir(case, output_substates = true)
```

14.219 Plot the reservoir solution

```
plot_reservoir(case.model, states, step = 135, key = :Saturations)
```

14.220 Load reference solution (OPM Flow)

We load a CSV file with the reference solution and set up plotting

```
csv_path = joinpath(egg_dir, "REFERENCE.CSV")
data, header = readdlm(csv_path, ',', header = true)
```

```
time_ref = data[:, 1]
time_jutul = deepcopy(ws.time)
wells = deepcopy(ws.wells)
wnames = collect(keys(wells))
nw = length(wnames)
day = si_unit(:day)
cmap = :tableau_hue_circle

inj = Symbol[]
prod = Symbol[]
for (wellname, well) in pairs(wells)
    qts = well[:wrat] + well[:orat]
    if sum(qts) > 0
        push!(inj, wellname)
    else
        push!(prod, wellname)
    end
end

function plot_well_comparison(response, well_names, reponse_name = "$response")
    fig = Figure(size = (1000, 400))
    if response == :bhp
        ys = 1/si_unit(:bar)
        yl = "Bottom hole pressure / Bar"
    elseif response == :wrat
        ys = si_unit(:day)
        yl = "Surface water rate / m³/day"
    elseif response == :orat
        ys = si_unit(:day)/(1000*si_unit(:stb))
        yl = "Surface oil rate / 10³ stb/day"
    else
        error("$response not ready.")
    end
    welltypes = []
    ax = Axis(fig[1:4, 1], xlabel = "Time / days", ylabel = yl)
    i = 1
    linehandles = []
    linelabels = []
    for well_name in well_names
        well = wells[well_name]
        label_in_csv = "$well_name:$response"
        ref_pos = findfirst(x -> x == label_in_csv, vec(header))
        qoi = copy(well[response]).*ys
        qoi_ref = data[:, ref_pos].*ys
        tot_rate = copy(well[:rate])
        @. qoi[tot_rate == 0] = NaN
        orat_ref = data[:, findfirst(x -> x == "$well_name:orat", vec(header))]
```

```
        wrat_ref = data[:, findfirst(x -> x == "$well_name:wrat", vec(header))]
        tot_rate_ref = orat_ref + wrat_ref
        @. qoi_ref[tot_rate_ref == 0] = NaN
        crange = (1, max(length(well_names), 2))
        lh = lines!(ax, time_jutul./day, abs.(qoi),
            color = i,
            colorrange = crange,
            label = "$well_name", colormap = cmap
        )
        push!(linehandles, lh)
        push!(linelabels, "$well_name")
        lines!(ax, time_ref./day, abs.(qoi_ref),
            color = i,
            colorrange = crange,
            linestyle = :dash,
            colormap = cmap
        )
        i += 1
    end
    l1 = LineElement(color = :black, linestyle = nothing)
    l2 = LineElement(color = :black, linestyle = :dash)

    Legend(fig[1:3, 2], linehandles, linelabels, nbanks = 3)
    Legend(fig[4, 2], [l1, l2], ["JutulDarcy.jl", "E100"])
    fig
end
```

14.221 Well responses and comparison

As the case is a two-phase model with water injection, we limit the results to plots of the producer water and oil rates. ### Water production rates

```
plot_well_comparison(:wrat, prod, "Producer water surface rate")
```

14.221.1 Oil production rates

```
plot_well_comparison(:orat, prod, "Producer oil surface rate")
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Comparison between JutulDarcy.jl and MRST This example contains validation of JutulDarcy.jl against MRST. In general, minor differences are observed. These can be traced back to a combination of different internal timestepping done by the simulators and that JutulDarcy by default uses a multisegment well formulation while MRST uses a standard instantaneous equilibrium model without well bore storage terms. These differences are most evident when simulators start up.

These cases have been exported using the MRST `jutul` module which can export MRST or Eclipse-type of cases to a JutulDarcy-compatible input format. They can then be simulated using simulate_mrst_case.

```
using JutulDarcy, Jutul
using GLMakie
```

14.222 Define a few utilities for plotting the MRST results

We are going to compare well responses against pre-computed results stored inside the JutulDarcy module.

```
function mrst_case_path(name)
    base_path, = splitdir(pathof(JutulDarcy))
    joinpath(base_path, "..", "test", "mrst", "$(name).mat")
end

function mrst_solution(result)
    return result.extra[:mrst]["extra"][1]["mrst_solution"]
end

function mrst_well_index(mrst_result, k)
    return findfirst(isequal("$k"), vec(mrst_result["names"]))
end

function get_mrst_comparison(wdata, ref, wname, t = :bhp)
    yscale = "m³/s"
    if t == :bhp
        tname = :bhp
        mname = "bhp"
        yscale = "Pa"
    elseif t == :qos
        tname = :orat
        mname = "qOs"
    elseif t == :qws
        tname = :wrat
        mname = "qWs"
    elseif t == :qgs
        tname = :grat
        mname = "qGs"
    else
        error("Not supported: $t")
    end
    jutul = wdata[tname]
    mrst = ref[mname][:, mrst_well_index(ref, wname)]

    return (jutul, mrst, tname, yscale)
end
```

```
function plot_comparison(wsol, ref, rep_t, t, wells_keys = keys(wsol.wells))
    fig = Figure()
    ax = Axis(fig[1, 1], xlabel = "time (days)")
    l = ""
    yscale = ""
    T = rep_t./(3600*24.0)
    for (w, d) in wsol.wells
        if !(w in wells_keys)
            continue
        end
        jutul, mrst, l, yscale = get_mrst_comparison(d, ref, w, t)
        lines!(ax, T, abs.(jutul), label = "$w")
        scatter!(ax, T, abs.(mrst), markersize = 8)
    end
    axislegend()
    ax.ylabel[] = "$l ($yscale)"
    fig
end
#-
```

14.223 The Egg model (oil-water compressible)

A two-phase model that is taken from the first member of the EGG ensemble. For more details, see the paper where the ensemble is introduced:

Jansen, Jan-Dirk, et al. “The egg model–a geological ensemble for reservoir simulation.” Geoscience Data Journal 1.2 (2014): 192-195. ### Simulate model

```
egg = simulate_mrst_case(mrst_case_path("egg"), info_level = -1)
wells = egg.wells
rep_t = egg.time
ref = mrst_solution(egg);
```

14.223.1 Compare well responses

```
injectors = [:INJECT1, :INJECT2, :INJECT3, :INJECT4, :INJECT5, :INJECT6, :INJECT7]
producers = [:PROD1, :PROD2, :PROD3, :PROD4]
#-
```

14.223.1.1 Bottom hole pressures

```
plot_comparison(wells, ref, rep_t, :bhp, injectors)
#-
```

14.223.1.2 Oil rates

```
plot_comparison(wells, ref, rep_t, :qos, producers)
#-
```

14.223.1.3 Water rates

```
plot_comparison(wells, ref, rep_t, :qws, producers)
```

14.224 SPE1 (black oil, disgas)

A shortened version of the SPE1 benchmark case.

Odeh, A.S. 1981. Comparison of Solutions to a Three-Dimensional Black-Oil Reservoir Simulation Problem. J Pet Technol 33 (1): 13–25. SPE-9723-PA

For comparison against other simulators, see the equivialent JutulSPE1 example in the Jutul module for MRST ### Simulate model

```
spe1 = simulate_mrst_case(mrst_case_path("spe1"), info_level = -1)
wells = spe1.wells
rep_t = spe1.time
ref = mrst_solution(spe1);
```

14.224.1 Compare well responses

14.224.1.1 Bottom hole pressures

```
#-
plot_comparison(wells, ref, rep_t, :bhp)
```

14.224.1.2 Gas rates

```
#-
plot_comparison(wells, ref, rep_t, :qgs, [:PRODUCER])
```

14.225 SPE3 (black oil, vapoil)

A black-oil variant of the SPE3 benchmark case.

Kenyon, D. “Third SPE comparative solution project: gas cycling of retrograde condensate reservoirs.” Journal of Petroleum Technology 39.08 (1987): 981-997 ### Simulate model

```
spe3 = simulate_mrst_case(mrst_case_path("spe3"), info_level = -1)
wells = spe3.wells
rep_t = spe3.time
ref = mrst_solution(spe3);
```

14.225.1 Compare well responses

14.225.1.1 Bottom hole pressures

```
plot_comparison(wells, ref, rep_t, :bhp, [:PRODUCER])
#-
#-
```

14.225.1.2 Gas rates

```
plot_comparison(wells, ref, rep_t, :qgs, [:PRODUCER])
#-
#-
```

14.225.1.3 Oil rates

```
plot_comparison(wells, ref, rep_t, :qos, [:PRODUCER])
#-
#-
```

14.226 SPE9 (black oil, disgas)

Example of the SPE9 model exported from MRST running in JutulDarcy.

[Killough, J. E. 1995. Ninth SPE comparative solution project: A reexamination of black-oil simulation. In SPE Reservoir Simulation Symposium, 12-15 February 1995, San Antonio, Texas. SPE 29110-MS] (http://dx.doi.org/10.2118/29110-MS)

For comparison against other simulators, see the equivialent JutulSPE9 example in the Jutul module for MRST ### Simulate model

```
spe9 = simulate_mrst_case(mrst_case_path("spe9"), info_level = -1)
wells = spe9.wells
rep_t = spe9.time
ref = mrst_solution(spe9);
```

14.226.1 Compare well responses

```
injectors = [:INJE1]
producers = [Symbol("PROD$i") for i in 1:25]
```

14.226.1.1 Injector water rate

```
plot_comparison(wells, ref, rep_t, :qws, injectors)
#-
```

14.226.1.2 Oil rates

```
plot_comparison(wells, ref, rep_t, :qos, producers)
#-
```

14.226.1.3 Water rates

```
plot_comparison(wells, ref, rep_t, :qws, producers)
#-
```

14.226.1.4 Bottom hole pressures

```
plot_comparison(wells, ref, rep_t, :bhp, producers)
#-
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Norne: Real field black-oil model The Norne model is a real field model. The model has been adapted so that the input file only contains features present in JutulDarcy, with the most notable omissions being removal of hysteresis and threshold pressures between equilibriation reqgions. For more details, see the OPM data webpage

```
using Jutul, JutulDarcy, GLMakie, DelimitedFiles, HYPRE, GeoEnergyIO
norne_dir = GeoEnergyIO.test_input_file_path("NORNE_NOHYST")
data_pth = joinpath(norne_dir, "NORNE_NOHYST.DATA")
data = parse_data_file(data_pth)
case = setup_case_from_data_file(data);
```

14.227 Unpack the case to see basic data structures

```
model = case.model
parameters = case.parameters
forces = case.forces
dt = case.dt;
```

14.228 Plot the reservoir mesh, wells and faults

We compose a few different plotting calls together to make a plot that shows the outline of the mesh, the fault structures and the well trajectories.

```
import Jutul: plot_mesh_edges!
reservoir = reservoir_domain(model)
mesh = physical_representation(reservoir)
wells = get_model_wells(model)
fig = Figure(size = (1200, 800))
ax = Axis3(fig[1, 1], zreversed = true)
```

```
plot_mesh_edges!(ax, mesh, alpha = 0.5)
for (k, w) in wells
    plot_well!(ax, mesh, w)
end
plot_faults!(ax, mesh, alpha = 0.5)
ax.azimuth[] = -3.0
ax.elevation[] = 0.5
fig
```

14.229 Plot the reservoir static properties in interactive viewer

```
fig = plot_reservoir(model, key = :porosity)
ax = fig.current_axis[]
plot_faults!(ax, mesh, alpha = 0.5)
ax.azimuth[] = -3.0
ax.elevation[] = 0.5
fig
```

14.230 Simulate the model

```
ws, states = simulate_reservoir(case, output_substates = true)
```

14.231 Plot the reservoir solution

```
fig = plot_reservoir(model, states, step = 247, key = :Saturations)
ax = fig.current_axis[]
ax.azimuth[] = -3.0
ax.elevation[] = 0.5
fig
```

14.232 Load reference and set up plotting

```
csv_path = joinpath(norne_dir, "REFERENCE.CSV")
data_ref, header = readdlm(csv_path, ',', header = true)
time_ref = data_ref[:, 1]
time_jutul = deepcopy(ws.time)
wells = deepcopy(ws.wells)
wnames = collect(keys(wells))
nw = length(wnames)
day = si_unit(:day)
cmap = :tableau_hue_circle

inj = Symbol[]
```

```
prod = Symbol[]
for (wellname, well) in pairs(wells)
    qts = well[:wrat] + well[:orat] + well[:grat]
    if sum(qts) > 0
        push!(inj, wellname)
    else
        push!(prod, wellname)
    end
end

function plot_well_comparison(response, well_names, reponse_name = "$response"; cumulative = fa
    fig = Figure(size = (1000, 400))
    if response == :bhp
        ys = 1/si_unit(:bar)
        yl = "Bottom hole pressure / Bar"
    elseif response == :wrat
        ys = si_unit(:day)
        if cumulative
            yl = "Cumulative water rate / m³"
        else
            yl = "Water rate / m³/day"
        end
    elseif response == :grat
        ys = si_unit(:day)/1e6
        if cumulative
            yl = "Cumulative gas rate / 10  m³"
        else
            yl = "Gas rate / 10  m³/day"
        end
    elseif response == :orat
        ys = si_unit(:day)/(1000*si_unit(:stb))
        if cumulative
            yl = "Cumulative oil rate / 10³ stb"
        else
            yl = "Oil rate / 10³ stb/day"
        end
    else
        error("$response not ready.")
    end
    welltypes = []
    ax = Axis(fig[1:4, 1], xlabel = "Time / days", ylabel = yl)
    i = 1
    linehandles = []
    linelabels = []
    for well_name in well_names
        well = wells[well_name]
        label_in_csv = "$well_name:$response"
```

```
        ref_pos = findfirst(x -> x == label_in_csv, vec(header))
        qoi = copy(well[response]).*ys
        qoi_ref = data_ref[:, ref_pos].*ys
        tot_rate = copy(well[:rate])
        grat_ref = data_ref[:, findfirst(x -> x == "$well_name:grat", vec(header))]
        orat_ref = data_ref[:, findfirst(x -> x == "$well_name:orat", vec(header))]
        wrat_ref = data_ref[:, findfirst(x -> x == "$well_name:wrat", vec(header))]
        tot_rate_ref = grat_ref + orat_ref + wrat_ref
        if cumulative
            @. qoi_ref[tot_rate_ref == 0] = 0
            @. qoi[tot_rate == 0] = 0
            qoi_ref = cumsum(qoi_ref.*diff([0, time_ref...]./day))
            qoi = cumsum(qoi.*diff([0, time_jutul...]./day))
        else
            @. qoi_ref[tot_rate_ref == 0] = NaN
            @. qoi[tot_rate == 0] = NaN
        end
        crange = (1, max(length(well_names), 2))
        lh = lines!(ax, time_jutul./day, abs.(qoi),
            color = i,
            colorrange = crange,
            label = "$well_name", colormap = cmap
        )
        push!(linehandles, lh)
        push!(linelabels, "$well_name")
        lines!(ax, time_ref./day, abs.(qoi_ref),
            color = i,
            colorrange = crange,
            linestyle = :dash,
            colormap = cmap
        )
        i += 1
    end
    l1 = LineElement(color = :black, linestyle = nothing)
    l2 = LineElement(color = :black, linestyle = :dash)

    Legend(fig[1:3, 2], linehandles, linelabels, nbanks = 3)
    Legend(fig[4, 2], [l1, l2], ["JutulDarcy.jl", "OPM Flow"])
    fig
end
```

14.233 Injector bhp

```
plot_well_comparison(:bhp, inj, "Bottom hole pressure")
```

14.234 Gas injection rates

14.234.1 Rates

```
plot_well_comparison(:grat, inj, "Gas surface injection rate")
```

14.235 Cumulative gas injection rates

```
plot_well_comparison(:grat, inj, "Cumulative gas surface injection rate", cumulative = true)
```

14.236 Water injection rates

14.236.1 Rates

```
plot_well_comparison(:wrat, inj, "Water surface injection rate")
```

14.236.2 Cumulative rates

```
plot_well_comparison(:wrat, inj, "Cumulative water surface injection rate", cumulative = true)
```

14.237 Producer bhp

```
plot_well_comparison(:bhp, prod, "Bottom hole pressure")
```

14.238 Oil production rates

14.238.1 Rates

```
plot_well_comparison(:orat, prod, "Oil surface production rate")
```

14.238.2 Cumulative rates

```
plot_well_comparison(:orat, prod, "Cumulative oil surface production rate", cumulative = true)
```

14.239 Gas production rates

14.239.1 Rates

```
plot_well_comparison(:grat, prod, "Gas surface production rate")
```

14.239.2 Cumulative rates

```
plot_well_comparison(:grat, prod, "Cumulative gas surface production rate", cumulative = true)
```

14.240 Water production rates

14.240.1 Rates

```
plot_well_comparison(:wrat, prod, "Water surface production rate")
```

14.240.2 Cumulative rates

```
plot_well_comparison(:wrat, prod, "Cumulative water surface production rate", cumulative = true
```

14.241 Interactive plotting of field statistics

```
plot_reservoir_measurables(case, ws, states, left = :fgpr, right = :pres)
```

14.242 Plot wells

```
plot_well_results(ws)
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # The OLYMPUS benchmark model: Two-phase corner-point reservoir Model from the ISAPP Optimization challenge

Two-phase complex corner-point model with primary and secondary production.

For more details, see olympus

```
using Jutul, JutulDarcy, GLMakie, DelimitedFiles, HYPRE
using Test # hide
olympus_dir = JutulDarcy.GeoEnergyIO.test_input_file_path("OLYMPUS_1")
case = setup_case_from_data_file(joinpath(olympus_dir, "OLYMPUS_1.DATA"), backend = :csr)
ws, states = simulate_reservoir(case, output_substates = true)
```

14.243 Plot the reservoir

```
plot_reservoir(case.model, key = :porosity)
```

14.244 Plot the saturations

```
plot_reservoir(case.model, states, step = 10, key = :Saturations)
```

14.245 Load reference and set up plotting

```
csv_path = joinpath(olympus_dir, "REFERENCE.CSV")
data, header = readdlm(csv_path, ',', header = true)
time_ref = data[:, 1]
time_jutul = deepcopy(ws.time)
wells = deepcopy(ws.wells)
wnames = collect(keys(wells))
nw = length(wnames)
day = si_unit(:day)
cmap = :tableau_hue_circle

inj = Symbol[]
prod = Symbol[]
for (wellname, well) in pairs(wells)
    qts = well[:wrat] + well[:orat]
    if sum(qts) > 0
        push!(inj, wellname)
    else
        push!(prod, wellname)
    end
end

function plot_well_comparison(response, well_names, reponse_name = "$response"; ylims = missing
    fig = Figure(size = (1000, 400))
    if response == :bhp
        ys = 1/si_unit(:bar)
        yl = "Bottom hole pressure / Bar"
    elseif response == :wrat
        ys = si_unit(:day)
        yl = "Surface water rate / m³/day"
    elseif response == :orat
        ys = si_unit(:day)/(1000*si_unit(:stb))
        yl = "Surface oil rate / 10³ stb/day"
    else
        error("$response not ready.")
    end
    welltypes = []
    ax = Axis(fig[1:4, 1], xlabel = "Time / days", ylabel = yl)
    i = 1
    linehandles = []
    linelabels = []
```

```
    for well_name in well_names
        well = wells[well_name]
        label_in_csv = "$well_name:$response"
        ref_pos = findfirst(x -> x == label_in_csv, vec(header))
        qoi = copy(well[response]).*ys
        qoi_ref = data[:, ref_pos].*ys

        val = sum(qoi.*diff([0; time_jutul]))       # hide
        val_ref = sum(qoi_ref.*diff([0; time_ref])) # hide
        @test abs(val - val_ref)/abs(val) < 0.03    # hide

        tot_rate = copy(well[:rate])
        @. qoi[tot_rate == 0] = NaN
        orat_ref = data[:, findfirst(x -> x == "$well_name:orat", vec(header))]
        wrat_ref = data[:, findfirst(x -> x == "$well_name:wrat", vec(header))]
        tot_rate_ref = orat_ref + wrat_ref
        @. qoi_ref[tot_rate_ref == 0] = NaN
        crange = (1, max(length(well_names), 2))
        lh = lines!(ax, time_jutul./day, abs.(qoi),
            color = i,
            colorrange = crange,
            label = "$well_name", colormap = cmap
        )
        push!(linehandles, lh)
        push!(linelabels, "$well_name")
        lines!(ax, time_ref./day, abs.(qoi_ref),
            color = i,
            colorrange = crange,
            linestyle = :dash,
            colormap = cmap
        )
        i += 1
        if !ismissing(ylims)
            ylims!(ax, ylims)
        end
    end
    l1 = LineElement(color = :black, linestyle = nothing)
    l2 = LineElement(color = :black, linestyle = :dash)

    Legend(fig[1:3, 2], linehandles, linelabels, nbanks = 3)
    Legend(fig[4, 2], [l1, l2], ["JutulDarcy.jl", "OPM Flow"])
    fig
end
```

14.246 Plot water production rates

```
plot_well_comparison(:wrat, prod, "Water surface production rate")
```

14.247 Plot oil production rates

```
plot_well_comparison(:orat, prod, "Oil surface production rate", ylims = (0, 5))
```

14.248 Plot water injection rates

```
plot_well_comparison(:wrat, inj, "Water injection rate")
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Polymer injection in a 2D black-oil reservoir model This example validates a small polymer model taken from the OPM-tests repository. The model is a 2D black-oil reservoir model with polymer injection. Adding polymer to the water phase increases the viscosity of the water and helps with mobility control. This is implemented in JutulDarcy as a tracer that can alter the properties of the system.

At present, the JutulDarcy polymer model supports the following features: - Polymer injection in the water phase - Varying degree of mixing between polymer and water - Adsorption of polymer to the rock - Polymer viscosity changes - Permeability reduction from polymer - Dead pore space for polymer part of water phase

Note that non-Newtonian rheology / shear effects are not yet implemented in the polymer model.

```
using GeoEnergyIO, Jutul, JutulDarcy, GLMakie, DelimitedFiles
pth = JutulDarcy.GeoEnergyIO.test_input_file_path("BOPOLYMER_NOSHEAR", "BOPOLYMER_NOSHEAR.DATA
data = parse_data_file(pth)
case = setup_case_from_data_file(data)
push!(case.model[:Reservoir].output_variables, :PolymerConcentration)
push!(case.model[:Reservoir].output_variables, :PhaseViscosities)
push!(case.model[:Reservoir].output_variables, :AdsorbedPolymerConcentration)

ws, states, time = simulate_reservoir(case)
```

14.249 Load the reference solution and set up plotting

```
ref_pth = JutulDarcy.GeoEnergyIO.test_input_file_path("BOPOLYMER_NOSHEAR", "result.txt")
tab, header = DelimitedFiles.readdlm(ref_pth, header = true)
header = vec(header)
units = tab[1, :]
tab = Float64.(tab[2:end, :])
getcol(x) = tab[:, findfirst(isequal(x), header)]
```

```
time_ref = getcol("TIME")

function plot_comparison(jutul, ref, label; pos = :rt)
    fig = Figure()
    ax = Axis(fig[1, 1]; xlabel = "Time / days", ylabel = label)
    lines!(ax, time_ref, ref, label = "Reference", color = :black)
    lines!(ax, time./si_unit(:day), jutul, label = "JutulDarcy", linestyle = :dash, linewidth =
    lines!(ax, [1285.0, 1285.0], [minimum([jutul; ref]), maximum([jutul; ref])], label = "Polym
    lines!(ax, [2960.0, 2960.0], [minimum([jutul; ref]), maximum([jutul; ref])], label = "Polym
    axislegend(position = pos)
    fig
end
```

14.250 Plot oil production rate

```
wopr_ref = getcol("WOPR:PROD01")
wopr = -ws[:PROD01, :orat]*si_unit(:day)
plot_comparison(wopr, wopr_ref, "Oil production rate / sm³/day")
```

14.251 Plot bottom hole pressure

The pressure required to inject water significantly increases as polymer is added. This is due to the increased viscosity of the water phase when polymer is part of the mixture.

```
wbhp_ref = getcol("WBHP:INJE01")
wbhp = ws[:INJE01, :bhp]./si_unit(:bar)
plot_comparison(wbhp, wbhp_ref, "Injector bottom hole pressure / bar", pos = :rb)
```

14.252 Plot the water front after polymer injection

```
reservoir = reservoir_domain(case.model)
g = physical_representation(reservoir)
```

14.253 Plot the water saturation front

```
fig, ax, plt = plot_cell_data(g, states[148][:Saturations][1, :])
ax.azimuth = 1.5
ax.elevation = 0
hidedecorations!(ax)
fig
```

14.254 Plot the polymer concentration

```
fig, ax, plt = plot_cell_data(g, states[148][:PolymerConcentration])
ax.azimuth = 1.5
ax.elevation = 0
hidedecorations!(ax)
fig
```

14.255 Plot the adsorbed polymer concentration

The polymer is adsorbed to the rock surface. This is a key part of the polymer model – the polymer is not only in the water phase but also adsorbed to the rock.

```
fig, ax, plt = plot_cell_data(g, states[148][:AdsorbedPolymerConcentration])
ax.azimuth = 1.5
ax.elevation = 0
hidedecorations!(ax)
fig
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # SPE1: Small black-oil gas injection Odeh, A.S. 1981. Comparison of Solutions to a Three-Dimensional Black-Oil Reservoir Simulation Problem. J Pet Technol 33 (1): 13–25. SPE-9723-PA

```
using Jutul, JutulDarcy, GLMakie, DelimitedFiles
spe1_dir = JutulDarcy.GeoEnergyIO.test_input_file_path("SPE1")
case = setup_case_from_data_file(joinpath(spe1_dir, "SPE1.DATA"))
ws, states = simulate_reservoir(case, output_substates = true)
```

Chapter 15

Load reference

```
csv_path = joinpath(spe1_dir, "REFERENCE.CSV")
data, header = readdlm(csv_path, ',', header = true)
time_ref = data[:, 1]
time_jutul = deepcopy(ws.time)
wells = deepcopy(ws.wells)
wnames = collect(keys(wells))
nw = length(wnames)
day = si_unit(:day)
cmap = :tableau_hue_circle

inj = Symbol[]
prod = Symbol[]
for (wellname, well) in pairs(wells)
    qts = well[:wrat] + well[:orat] + well[:grat]
    if sum(qts) > 0
        push!(inj, wellname)
    else
        push!(prod, wellname)
    end
end

function plot_well_comparison(response, well_names, reponse_name = "$response")
    fig = Figure(size = (1000, 400))
    if response == :bhp
        ys = 1/si_unit(:bar)
        yl = "Bottom hole pressure / Bar"
    elseif response == :wrat
        ys = si_unit(:day)
        yl = "Surface water rate / m³/day"
    elseif response == :grat
        ys = si_unit(:day)/1e6
        yl = "Surface gas rate / 10  m³/day"
```

```
    elseif response == :orat
        ys = si_unit(:day)/(1000*si_unit(:stb))
        yl = "Surface oil rate / 10³ stb/day"
    else
        error("$response not ready.")
    end
    welltypes = []
    ax = Axis(fig[1:4, 1], xlabel = "Time / days", ylabel = yl)
    i = 1
    linehandles = []
    linelabels = []
    for well_name in well_names
        well = wells[well_name]
        label_in_csv = "$well_name:$response"
        ref_pos = findfirst(x -> x == label_in_csv, vec(header))
        qoi = copy(well[response]).*ys
        qoi_ref = data[:, ref_pos].*ys
        tot_rate = copy(well[:rate])
        @. qoi[tot_rate == 0] = NaN
        grat_ref = data[:, findfirst(x -> x == "$well_name:grat", vec(header))]
        orat_ref = data[:, findfirst(x -> x == "$well_name:orat", vec(header))]
        wrat_ref = data[:, findfirst(x -> x == "$well_name:wrat", vec(header))]
        tot_rate_ref = grat_ref + orat_ref + wrat_ref
        @. qoi_ref[tot_rate_ref == 0] = NaN
        crange = (1, max(length(well_names), 2))
        lh = lines!(ax, time_jutul./day, abs.(qoi),
            color = i,
            colorrange = crange,
            label = "$well_name", colormap = cmap
        )
        push!(linehandles, lh)
        push!(linelabels, "$well_name")
        lines!(ax, time_ref./day, abs.(qoi_ref),
            color = i,
            colorrange = crange,
            linestyle = :dash,
            colormap = cmap
        )
        i += 1
    end
    l1 = LineElement(color = :black, linestyle = nothing)
    l2 = LineElement(color = :black, linestyle = :dash)

    Legend(fig[1:3, 2], linehandles, linelabels, nbanks = 3)
    Legend(fig[4, 2], [l1, l2], ["JutulDarcy.jl", "OPM Flow"])
    fig
end
```

15.1 Injector BHP

```
plot_well_comparison(:bhp, inj, "Bottom hole pressure")
```

15.2 Producer BHP

```
plot_well_comparison(:bhp, prod, "Bottom hole pressure")
```

15.3 Producer oil rate

```
plot_well_comparison(:orat, prod, "Oil surface rate")
```

15.4 Producer gas rate

```
plot_well_comparison(:grat, prod, "Gas surface rate")
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # SPE9: Black-oil depletion with dissolved gas [Killough, J. E. 1995. Ninth SPE comparative solution project: A reexamination of black-oil simulation. In SPE Reservoir Simulation Symposium, 12-15 February 1995, San Antonio, Texas. SPE 29110-MS] (http://dx.doi.org/10.2118/29110-MS) spe9

```
using Jutul, JutulDarcy, GLMakie, DelimitedFiles
spe9_dir = JutulDarcy.GeoEnergyIO.test_input_file_path("SPE9")
case = setup_case_from_data_file(joinpath(spe9_dir, "SPE9.DATA"))
ws, states = simulate_reservoir(case, output_substates = true)
##
csv_path = joinpath(spe9_dir, "REFERENCE.CSV")
data, header = readdlm(csv_path, ',', header = true)
##
time_ref = data[:, 1]
time_jutul = deepcopy(ws.time)
wells = deepcopy(ws.wells)
wnames = collect(keys(wells))
nw = length(wnames)
day = si_unit(:day)
cmap = :tableau_hue_circle

inj = Symbol[]
prod = Symbol[]
for (wellname, well) in pairs(wells)
    qts = well[:wrat] + well[:orat] + well[:grat]
    if sum(qts) > 0
```

```
        push!(inj, wellname)
    else
        push!(prod, wellname)
    end
end

function plot_well_comparison(response, well_names, reponse_name = "$response")
    fig = Figure(size = (1000, 400))
    if response == :bhp
        ys = 1/si_unit(:bar)
        yl = "Bottom hole pressure / Bar"
    elseif response == :wrat
        ys = si_unit(:day)
        yl = "Surface water rate / m³/day"
    elseif response == :grat
        ys = si_unit(:day)/1e6
        yl = "Surface gas rate / 10  m³/day"
    elseif response == :orat
        ys = si_unit(:day)/(1000*si_unit(:stb))
        yl = "Surface oil rate / 10³ stb/day"
    else
        error("$response not ready.")
    end
    welltypes = []
    ax = Axis(fig[1:4, 1], xlabel = "Time / days", ylabel = yl)
    i = 1
    linehandles = []
    linelabels = []
    for well_name in well_names
        well = wells[well_name]
        label_in_csv = "$well_name:$response"
        ref_pos = findfirst(x -> x == label_in_csv, vec(header))
        qoi = copy(well[response]).*ys
        qoi_ref = data[:, ref_pos].*ys
        tot_rate = copy(well[:rate])
        @. qoi[tot_rate == 0] = NaN
        grat_ref = data[:, findfirst(x -> x == "$well_name:grat", vec(header))]
        orat_ref = data[:, findfirst(x -> x == "$well_name:orat", vec(header))]
        wrat_ref = data[:, findfirst(x -> x == "$well_name:wrat", vec(header))]
        tot_rate_ref = grat_ref + orat_ref + wrat_ref
        @. qoi_ref[tot_rate_ref == 0] = NaN
        crange = (1, max(length(well_names), 2))
        lh = lines!(ax, time_jutul./day, abs.(qoi),
            color = i,
            colorrange = crange,
            label = "$well_name", colormap = cmap
        )
```

```
        push!(linehandles, lh)
        push!(linelabels, "$well_name")
        lines!(ax, time_ref./day, abs.(qoi_ref),
            color = i,
            colorrange = crange,
            linestyle = :dash,
            colormap = cmap
        )
        i += 1
    end
    l1 = LineElement(color = :black, linestyle = nothing)
    l2 = LineElement(color = :black, linestyle = :dash)

    Legend(fig[1:3, 2], linehandles, linelabels, nbanks = 3)
    Legend(fig[4, 2], [l1, l2], ["JutulDarcy.jl", "E100"])
    fig
end
```

15.5 Producer BHP

```
plot_well_comparison(:bhp, prod, "Bottom hole pressure")
```

15.6 Producer water rate

```
plot_well_comparison(:wrat, prod, "Water surface rate")
```

15.7 Injector water rate

```
plot_well_comparison(:wrat, inj, "Water surface rate")
```

15.8 Oil production rate

```
plot_well_comparison(:orat, prod, "Oil surface rate")
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation. # Aquifer thermal energy storage (ATES) validation This example validates JutulDarcy's thermal solver against results from a commercial simulator. The test case is a simple ATES model with a single pair of wells (hot / cold) where the cold well is used for pressure support with a mirrored injection rate. Towards the later part of the schedule, the cold well reinjects water that is a higher temperature than the background.

This case is completely specified in the `ATES_TEST.DATA` file which was provided by TNO. The model is a structured mesh with 472 500 active cells.

```
using Jutul, JutulDarcy, GeoEnergyIO, DelimitedFiles, HYPRE, GLMakie
basepth = GeoEnergyIO.test_input_file_path("ATES_TEST")
data = parse_data_file(joinpath(basepth, "ATES_TEST.DATA"))

wdata, wheader = readdlm(joinpath(basepth, "wells.txt"), ',', header = true)
cdata, cheader = readdlm(joinpath(basepth, "cells.txt"), ',', header = true)
case = setup_case_from_data_file(data)
ws, states, t_seconds = simulate_reservoir(case, info_level = 1);
```

15.9 Plot the reservoir and monitor points

We will monitor points close to the warm and cold wells for comparison with a commercial simulator. These can be identified by their IJK triplets, and we plot these in orange and blue.

```
reservoir = reservoir_domain(case)
G = physical_representation(reservoir)

cell_warm1 = cell_index(G, (105,75,6))
cell_warm2 = cell_index(G, (106,75,6))

cell_cold1 = cell_index(G, (50,75,6))
cell_cold2 = cell_index(G, (51,75,6))

fig = Figure(size = (800, 800))
ax = Axis3(fig[1, 1], zreversed = true, azimuth = 4.55, elevation = 0.2)
for (wname, w) in get_model_wells(case)
    plot_well!(ax, G, w)
end
Jutul.plot_mesh_edges!(ax, G, alpha = 0.1)
plot_mesh!(ax, G, color = :orange, cells = [cell_warm1, cell_warm2])
plot_mesh!(ax, G, color = :blue, cells = [cell_cold1, cell_cold2])
fig
```

15.10 Plot the permeability

The model contains a high permeable aquifer layer in the middle of the model. The high permeable layer conducts the flow between the wells, and the low permeable layers above and below the aquifer layer conduct heat.

```
fig = Figure(size = (800, 800))
ax = Axis3(fig[1, 1], zreversed = true, azimuth = 4.55, elevation = 0.2)
for (wname, w) in get_model_wells(case)
    plot_well!(ax, G, w)
end
plt = plot_cell_data!(ax, G, reservoir[:permeability][1, :]./si_unit(:darcy),
```

```
    shading = NoShading,
    colormap = :thermal
)
Colorbar(fig[2, 1], plt, label = "Horizontal permeability (darcy)", vertical = false)
fig
```

15.11 Plot the water rate in the wells

The water rate in the wells is shown below. The well rates are mirrored in that the cold well injects the same amount of water as the warm well produces and vice versa. Initially, the warm well injects water and the cold well produces to maintain aquifer pressure. Later on, the warm well produces warm water and the cold well injects utilized cold water at a slightly higher temperature than that of the reservoir to balance the pressure.

```
day = si_unit(:day)
t_jutul = t_seconds./day
wrat_cold = ws[:COLD][:wrat]*day
wrat_warm = ws[:WARM][:wrat]*day
fig = Figure()
ax = Axis(fig[1, 1], xlabel = "Time elapsed (days)", ylabel = "Water rate (m³/day)", title = "
lines!(ax, t_jutul, wrat_cold, label = "COLD well", color = :blue)
lines!(ax, t_jutul, wrat_warm, label = "WARM well", color = :orange)
axislegend(position = :ct)
fig
```

15.12 Plot the final temperature in the reservoir

We see the final temperature distribution in the reservoir. The regions near both wells are warmer than the rest of the reservoir, with the hot well being the warmest.

```
fig = Figure(size = (800, 800))
ax = Axis3(fig[1, 1], zreversed = true, azimuth = 4.55, elevation = 0.2)
for (wname, w) in get_model_wells(case)
    plot_well!(ax, G, w)
end
Jutul.plot_mesh_edges!(ax, G, alpha = 0.1)
temp = states[end][:Temperature] .- 273.15
plt = plot_cell_data!(ax, G, temp,
    colormap = :thermal,
    cells = findall(x -> x > 18, temp),
    transparency = true,
    shading = NoShading
)
Colorbar(fig[2, 1], plt, label = "Temperature (°C)", vertical = false)
fig
```

15.13 Plot the temperature near the warm well and compare to E300

We compare the temperature in cells close to the warm well in JutulDarcy and the same case simulated in E300, demonstrating excellent agreement.

```
warm1 = map(x -> x[:Temperature][cell_warm1] - 273.15, states)
warm2 = map(x -> x[:Temperature][cell_warm2] - 273.15, states)

t_e300 = cdata[:, 1]
warm1_e300 = cdata[:, 2]
warm2_e300 = cdata[:, 3]

fig = Figure()
ax = Axis(fig[1, 1], xlabel = "Time elapsed (days)", ylabel = "Temperature", title = "Temperatu
lines!(ax, t_jutul, warm1, label = "JutulDarcy, cell (105,75,6)", color = :orange)
lines!(ax, t_e300, warm1_e300, label = "E300, cell (105,75,6)", linestyle = :dash, linewidth =

lines!(ax, t_jutul, warm2, label = "JutulDarcy, cell (106,75,6)", color = :blue)
lines!(ax, t_e300, warm2_e300, label = "E300, cell (106,75,6)", linestyle = :dash, linewidth =
axislegend(position = :cb)
fig
```

15.14 Plot the temperature near the cold well and compare

We note a similar match between the solvers near the cold well.

```
cold1 = map(x -> x[:Temperature][cell_cold1] - 273.15, states)
cold2 = map(x -> x[:Temperature][cell_cold2] - 273.15, states)

t_e300 = cdata[:, 1]
cold1_e300 = cdata[:, 4]
cold2_e300 = cdata[:, 5]

fig = Figure()
ax = Axis(fig[1, 1], xlabel = "Time elapsed (days)", ylabel = "Temperature", title = "Temperatu
lines!(ax, t_jutul, cold1, label = "JutulDarcy, cell (50,75,6)", color = :orange)
lines!(ax, t_e300, cold1_e300, label = "E300, cell (50,75,6)", linestyle = :dash, linewidth = 3
lines!(ax, t_jutul, cold2, label = "JutulDarcy, cell (51,75,6)", color = :blue)
lines!(ax, t_e300, cold2_e300, label = "E300, cell (51,75,6)", linestyle = :dash, linewidth = 3
axislegend(position = :ct)
fig
```

15.15 Plot the well temperatures

Finally, we compare the reported temperatures in the wells between JutulDarcy and E300. These values are a mix of prescribed conditions (during injection) and the solution values (during produc-

tion).

```
t_well_e300 = wdata[:, 1]
warm_e300 = wdata[:, 2]
warm_jutul = ws[:WARM][:temperature] .- 273.15
cold_e300 = wdata[:, 3]
cold_jutul = ws[:COLD][:temperature] .- 273.15

fig = Figure()
ax = Axis(fig[1, 1], xlabel = "Time elapsed (days)", ylabel = "Temperature (°C)", title = "Wel
lines!(ax, t_jutul, warm_jutul, label = "JutulDarcy, WARM", color = :orange)
lines!(ax, t_well_e300, warm_e300, label = "E300, WARM", linestyle = :dash, linewidth = 3, colo
lines!(ax, t_jutul, cold_jutul, label = "JutulDarcy, COLD", color = :blue)
lines!(ax, t_well_e300, cold_e300, label = "E300, COLD", linestyle = :dash, linewidth = 3, colo
axislegend(position = :cb)
fig
```

15.16 Plot the total energy in the reservoir

We plot the total energy in the reservoir relative to the initial condition by summing up the thermal energy in all cells at the initial state and using this as the baseline. The total energy in the reservoir matches the injected and produced energy, as there are no open boundary conditions.

```
E0 = sum(states[1][:TotalThermalEnergy])
energy = map(x -> sum(x[:TotalThermalEnergy]) - E0, states)
lines(t_jutul, energy, axis = (xlabel = "Time elapsed (days)", ylabel = "Total energy (J)", ti
```

15.17 Plot the reservoir in the interactive viewer

If you are running this example yourself, you can launch an interactive viewer and explore the evolution of the model.

```
plot_reservoir(case, states, key = :Pressure, step = 100)
```

Note: This is a simplified version for the PDF documentation. For the full interactive example with code execution, plots, and detailed output, please visit the online documentation.