JutulDarcy.jl Documentation

Olav Møyner and contributors

2025-11-01

Contents

Chapter 1

JutulDarcy.jl

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/")
```

```
generate_jutuldarcy_examples
```

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.

```
@example intro_ex using JutulDarcy, Jutul Darcy, bar, kg, meter, day = si_units(:darcy,
:bar, :kilogram, :meter, :day) 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) # Show the properties in the
domain domain
```

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.

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

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.

```
@example intro_ex model = setup_reservoir_model(domain, sys, wells = [Injector, Producer])
```

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:

```
@example intro_ex reservoir_model(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 and the phase density functions by constant compressibilities:

```
@example intro_ex c = [1e-6, 1e-4]/bar density = ConstantCompressibilityDensities( p_ref = 100*bar,     density_ref = reference_densities,     compressibility = c ) kr = BrooksCoreyRelativePermeabilities(sys, [2.0, 3.0]) replace_variables!(model, PhaseMassDensities = density, RelativePermeabilities = kr) nothing #hide
```

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.

```
@example intro_ex state0 = setup_reservoir_state(model,     Pressure = 120bar, Saturations = [1.0, 0.0] )
```

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.

```
@example intro_ex nstep = 25 dt = fill(365.0day, nstep);
```

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.

```
@example intro_ex parameters = setup_parameters(model) pv = pore_volume(model, parameters) inj_rate = 1.5*sum(pv)/sum(dt) rate_target = TotalRateTarget(inj_rate)
```

The producer is set to operate at a fixed pressure:

```
@example intro_ex bhp_target = BottomHolePressureTarget(100bar)
```

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.

```
@example intro_ex I_ctrl = InjectorControl(rate_target, [0.0, 1.0], density =
rhoGS) P_ctrl = ProducerControl(bhp_target) controls = Dict(:Injector => I_ctrl,
:Producer => P_ctrl) forces = setup_reservoir_forces(model, control = controls);
```

2.6 Simulate and analyze results

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

```
@example intro_ex wd, states, t = simulate_reservoir(state0, model, dt, parameters
= parameters, forces = forces)
```

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

```
@example intro_ex wd(:Producer)
```

Let us look at the pressure evolution in the injector:

```
@example intro_ex wd(:Injector, :bhp)
```

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

```
@example intro_ex using GLMakie grat = wd[:Producer, :grat] lrat = wd[:Producer,
:lrat] bhp = wd[:Injector, :bhp] fig = Figure(size = (1200, 400)) ax = Axis(fig[1,
1],     title = "Injector",     xlabel = "Time / days",     ylabel = "Bottom hole
pressure / bar") lines!(ax, t/day, bhp./bar) ax = Axis(fig[1, 2],     title =
"Producer",     xlabel = "Time / days",     ylabel = "Production rate / m³/day")
lines!(ax, t/day, abs.(grat).*day) lines!(ax, t/day, abs.(lrat).*day) fig
```

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

```
@example intro_ex plot_reservoir(model, states, key = :Saturations, step = 3)   #
High-level API
```

2.7 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!

2.7.1 Meshes

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

2.7.2 Reservoir

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

`reservoir_domain`
`get_1d_reservoir`

2.7.3 Wells

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

`setup_well`
`setup_vertical_well`

2.7.4 Model

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

`setup_reservoir_model`

2.7.5 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.

`setup_reservoir_state`
`EquilibriumRegion`
`JutulDarcy.equilibriate_state`

2.8 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.

`simulate_reservoir`
`setup_reservoir_simulator`
`ReservoirSimResult`

Chapter 3

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`)

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

3.1.1 Simulation of .MAT files

```
setup_case_from_mrst
simulate_mrst_case
```

3.1.2 MRST-specific types

```
Jutul.MRSTWrapMesh
```

3.2 DATA-files from commercial reservoir modelling software

JutulDarcy can set up cases from Eclipse-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:

```
simulate_data_file
setup_case_from_data_file
JutulDarcy.setup_case_from_parsed_data
JutulDarcy.convert_co2store_to_co2_brine
```

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. # 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.

3.3 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, \c
```

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`

3.3.1 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.

```
LiquidPhase
AqueousPhase
VaporPhase
```

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

`JutulDarcy.MultiPhaseSystem`

3.3.2 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!`.

`JutulDarcy.TotalMasses`
`JutulDarcy.update_total_masses!`
`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.

`Jutul.apply_forces_to_equation!`
`Jutul.update_cross_term_in_entity!`

3.4 Single-phase flow

`SinglePhaseSystem`

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
```

`\rho` is the phase mass density and `\phi` 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 `\nabla 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, `\mu` 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.

3.5 Multi-phase, immiscible flow

`ImmiscibleSystem`

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 `\alpha` as `S_\alpha`. 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
```

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 \ge S_\alpha \ge 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) - \
r_w = \frac{\partial}{\partial t} (S_w \rho_w \phi) + \nabla \cdot (\rho_w \vec{v}_w) - \rho_w
```

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 \n
```

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_\alpha that account for capillary pressure, e.g. p_w = p_n + p_c(S_w).

3.5.1 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.

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

StandardBlackOilSystem

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
```

and gas,

```
r_g = \rho_g^s \left( \frac{\partial}{\partial t}( (b_g S_g + R_s b_o S_o) \phi) + \nabla \cdo
```

The model uses the notion of surface (or reference densities) `\rho_o^s, \rho_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.

3.7 Compositional: Multi-phase, multi-component flow

`MultiPhaseCompositionalSystemLV`

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) + \nabl
```

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`.

3.8 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}\Bigl(\rho_r U_r (1-\phi) + \sum_\alpha \rho_\alpha S_\alpha U
```

```
JutulDarcy.add_thermal_to_model!
```

Chapter 4

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.

4.1 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{\partia
```

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

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

Here, `\omega` 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}
```

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.

4.2 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.

4.2.1 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.

4.2.2 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.

`reservoir_linsolve`

4.2.2.1 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
   \left(\frac{\partial r_{w}}{\partial p}\right)_{ij} & \left(\frac{\partial r_{w}}{\partial s
   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.

4.2.2.1.1 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.

`CPRPreconditioner`

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) - \
r_w = \frac{\partial}{\partial t} (S_w \rho_w \phi) + \nabla \cdot (\rho_w \vec{v}_w) - \rho_w
```

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}{\
= \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) \nabla p \\
= - \nabla \mathbf{K}\lambda_t \nabla p = 0
```

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 - J P(\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)`

4.2.2.2 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}\ma
```

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

JutulDarcy.setup_reservoir_forces

4.3 Source terms

SourceTerm
JutulDarcy.FlowSourceType

4.4 Boundary conditions

FlowBoundaryCondition
flow_boundary_condition

Chapter 5

Wells and controls

5.1 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.

`JutulDarcy.WellGroup`

5.2 Types of wells

5.2.1 Simple wells

`JutulDarcy.SimpleWell`

5.2.1.1 Equations

5.2.2 Multisegment wells

`WellDomain`
`MultiSegmentWell`
`JutulDarcy.SegmentWellBoreFrictionHB`
`JutulDarcy.PotentialDropBalanceWell`
`MixedWellSegmentFlow`

5.3 Well controls and limits

5.3.1 Types of well controls

`InjectorControl`
`ProducerControl`
`DisabledControl`

`JutulDarcy.replace_target`
`JutulDarcy.default_limits`

5.3.2 Types of well targets

`BottomHolePressureTarget`
`SinglePhaseRateTarget`
`SurfaceLiquidRateTarget`
`SurfaceOilRateTarget`
`SurfaceGasRateTarget`
`SurfaceWaterRateTarget`
`TotalRateTarget`
`HistoricalReservoirVoidageTarget`
`ReservoirVoidageTarget`
`DisabledTarget`
`JutulDarcy.TotalReservoirRateTarget`
`TotalMassRateTarget`

5.3.3 Implementation of well controls

`JutulDarcy.well_target`

5.3.4 Well outputs

`JutulDarcy.print_well_result_table`

5.3.5 Imposing limits on wells (multiple constraints)

5.4 Well forces

5.4.1 Perforations and WI adjustments

`PerforationMask`
`JutulDarcy.Perforations`
`compute_peaceman_index`

5.4.2 Other forces

Can use `SourceTerm` or `FlowBoundaryCondition` # Primary variables

5.5 Fluid systems

5.5.1 General

`Pressure`
`ImmiscibleSaturation`

5.5.2 Immiscible flow

`Saturations`

5.5.3 Black-oil flow

`BlackOilUnknown`
`BlackOilX`

5.5.4 Compositional flow

`OverallMoleFractions`

5.6 Wells

`TotalMassFlux`

5.7 WellGroup / Facility

`TotalSurfaceMassRate`

Chapter 6

Secondary variables (properties)

6.1 Fluid systems

6.1.1 General

6.1.1.1 Relative permeabilities

`BrooksCoreyRelativePermeabilities`
`RelativePermeabilities`
`JutulDarcy.TabulatedSimpleRelativePermeabilities`
`JutulDarcy.ReservoirRelativePermeabilities`

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

`JutulDarcy.NoHysteresis`
`JutulDarcy.CarlsonHysteresis`
`JutulDarcy.KilloughHysteresis`
`JutulDarcy.JargonHysteresis`
`JutulDarcy.ImbibitionOnlyHysteresis`

`PhaseRelativePermeability`

6.1.1.2 Phase viscosities

`DeckPhaseViscosities`
`JutulDarcy.ConstMuBTable`
`JutulDarcy.MuBTable`

`PhaseMassDensities`

6.1.1.3 Phase densities

`JutulDarcy.DeckPhaseMassDensities`

6.1.1.4 Shrinkage factors

`DeckShrinkageFactors`

ConstantCompressibilityDensities

6.1.2 Black-oil flow

6.1.3 Compositional flow

PhaseMassFractions
JutulDarcy.KValueWrapper

6.2 Wells

TotalMass

Chapter 7

Parameters

7.1 General

`JutulDarcy.FluidVolume`

7.2 Reservoir parameters

7.2.1 Transmissibility

`JutulDarcy.Transmissibilities`

`JutulDarcy.reservoir_transmissibility`

7.2.2 Other

`JutulDarcy.TwoPointGravityDifference`
`JutulDarcy.ConnateWater`
`JutulDarcy.EndPointScalingCoefficients`

7.3 Well parameters

`JutulDarcy.WellIndices`
`JutulDarcy.PerforationGravityDifference`

7.4 Thermal

`JutulDarcy.FluidThermalConductivities`

Chapter 8

Plotting and visualization

```
plot_reservoir
plot_well_results
plot_reservoir_measurables
JutulDarcy.plot_reservoir_simulation_result
JutulDarcy.plot_well!
```

Chapter 9

Utilities

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

9.1 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
```

```
JutulDarcy.CO2Properties.compute_co2_brine_props
JutulDarcy.CO2Properties.pvt_brine_RoweChou1970
JutulDarcy.CO2Properties.activity_co2_DS2003
JutulDarcy.CO2Properties.viscosity_brine_co2_mixture_IC2012
JutulDarcy.CO2Properties.pvt_brine_BatzleWang1992
JutulDarcy.CO2Properties.viscosity_co2_Fenghour1998
JutulDarcy.CO2Properties.pvt_co2_RedlichKwong1949
JutulDarcy.CO2Properties.viscosity_gas_mixture_Davidson1993
```

9.2 Relative permeability functions

```
JutulDarcy.brooks_corey_relperm
```

9.3 CO2 inventory

```
JutulDarcy.co2_inventory
JutulDarcy.plot_co2_inventory
```

9.4 API utilities

```
reservoir_model
JutulDarcy.reservoir_storage
JutulDarcy.well_symbols
```

9.5 Model reduction

```
coarsen_reservoir_case
```

9.6 Adjoints and gradients

```
JutulDarcy.reservoir_sensitivities
JutulDarcy.well_mismatch
Jutul.LBFGS.unit_box_bfgs
```

9.7 Well outputs

```
get_model_wells
well_output
full_well_outputs
```

Chapter 10

Package docstring

JutulDarcy

Chapter 11

Multi-threading and MPI support

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

11.1 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.

11.1.1 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.

11.1.2 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.

11.1.3 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.

11.1.4 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

11.2 Solving with MPI in practice

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

11.2.1 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.

11.2.2 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:

`JutulDarcy.simulate_reservoir_parray`

11.2.3 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).

11.2.4 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.

11.3 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]
```

11.3.1 Running on CPU

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

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

11.3.2 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);
```

11.3.3 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.

11.4 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. # 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.

11.5 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.

11.5.1 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.

11.5.2 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.

11.5.3 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.

11.5.4 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.

11.6 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.

11.6.1 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. # 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.

11.7 Input and output

11.7.1 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.

11.7.2 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.

11.7.3 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.

11.7.4 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.

11.7.5 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.

11.7.6 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).

11.8 Plotting

11.8.1 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

11.9 Miscellaneous

11.9.1 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.

11.9.2 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.

11.9.3 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.

11.9.4 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.

11.9.5 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. # 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.

11.10 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)

11.11 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)

11.12 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)

11.13 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)

Chapter 12

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:

```
Modules = [Jutul, Jutul.ConvergenceMonitors]
```