Computational Geosciences manuscript No.
(will be inserted by the editor)

Fully Implicit Simulation of Polymer Flooding with MRST

Kai Bao · Knut-Andreas Lie · Olav Møyner · Ming Liu

Received: date / Accepted: date

Abstract The present work describes a fully implicit simulator for polymer injection implemented in the free, open-source MATLAB Reservoir Simulation Toolbox (MRST). Polymer injection is one of the widely used enhanced oil recovery (EOR) techniques and complicated physical process is involved, which makes accurate simulation very challenging. The proposed work is intended for providing a powerful and flexible tool to investigate the polymer injection process in realistic reservoir scenarios.

Within the model, the polymer component is assumed to be only transported in the water phase and adsorbed in the rock. The hydrocarbon phases are not influenced by the polymer and they are described with the standard, three-phase, black-oil equations. The effects of the polymer are simulated based on the Todd–Longstaff mixing model, accounting for adsorption, inaccessible pore space, and permeability reduction effects. Shear-thinning/thickening effects based on shear rate are also included by the means of a separate inner-Newton iteration process within the global nonlinear iteration. The implementation is based on the automatic differentiation framework in MRST (MRST-AD), and an iterative linear solver with a constrained pressure residual (CPR) preconditioner is used to solve the resulting linear systems efficiently.

We discuss certain implementation details to show how convenient it is to use the existing functionality in MRST to develop an accurate and efficient polymer flooding simulator for real fields. With its modular design, vectorized implementation, support for stratigraphic and general unstructured grids, and automatic differentiation framework, MRST is a very powerful prototyping and experimentation platform for development of new reservoir simulators.

Kai Bao, Knut-Andreas Lie, Olav Møyner
Department of Applied Mathmatics, SINTEF ICT
P.O. Box 124 Blindern, 0314 Oslo, Norway
E-mail: {Kai.Bao, Knut-Andreas.Lie, Olav.Moyner}@sintef.no

Ming Liu
Statoil ASA
28^{th} Floor, West Tower, Twin Towers, B-12, Jianwai Ave.
Chaoyang District, Beijing 100022, P.R.C.
E-mail: miliu@statoil.com

To verify the simulator, we first compare it with a commercial simulator and good agreement is achieved. Then, we apply the new simulator to a few realistic reservoir models to investigate the effect of adding polymer injection and computational efficiency is demonstrated. Finally, we combine existing optimization functionality in MRST with the new polymer simulator to optimize polymer flooding for two different reservoir models. We argue that the presented software framework can be used as an efficient prototyping tool to evaluate new models for polymer-water-flooding processes in real reservoir fields.

Keywords MRST · Open-source implementation · Polymer flooding · Black-oil · Flow diagnostics

1 Introduction

Water-based methods for enhanced oil recovery (EOR) consist of adding active chemical or biological substances that modify the physical properties of the fluids and/or the porous media at the interface between oil and water [7]. Polymer flooding is one of the most widely applied water-based EOR techniques [23]. In polymer flooding, polymer molecules of relatively large size are added to the injected water to reduce its mobility and hence improve the local displacement and the volumetric sweep efficiency of the waterflood [13, 7, 24]. The most important mechanism is that the dissolved polymer molecules increase the brine viscosity, which increases the saturation behind the water front, enables the water drive to push more oil through the reservoir and reduces its tendency of channeling through high-flow zones. The presence of polymer may also reduce the permeability of the reservoir rock. Onshore, polymer flooding can be considered a mature technology, having migrated from USA to China where the world's largest polymer-driven oil production is found in the Daqing Oil field. Offshore applications are few and more challenging because of high-salinity formation water, well placement and large well spacing, stability under injection, produced water and polymer treatment and other HSE (health, safety, and environment) requirements, logistic difficulties, etc.

In its most basic form, polymer flooding is described by a flow model that consists of two or three phases and three or four fluid components. Compared with the standard black-oil models, the presence of long-chain polymer molecules in the water phase introduces a series of new flow effects. Depending on the types of the polymer used, and also the rock and brine properties, polymer can be adsorbed onto the surface of the reservoir rock, and contribute to reducing porosity and permeability. Polymer flooding is in reality a miscible process, but is typically simulated on a field scale using immiscible flow models which use empirical mixture models to account for unresolved miscibility effects. Moreover, the diluted polymer solution is in most cases pseudoplastic or shear-thinning, and hence has lower viscosity near injection wells and other high-flow zones where shear rates are high. This non-Newtonian fluid rheology improves injectivity and gradually introduces the desired mobility control in terms of a stronger displacement front, but may also reduce sweep efficiency since the polymer solution will have a higher tendency of flowing through high-permeable regions. Polymer solutions can also exhibit pseudodilatant or shear-thickening behavior, which improves sweep efficiency and reduces injectivity. Understanding and being able to accurately simulate the rheological behavior of the polymer-water mixture on a reservoir scale

is therefore important to design successful polymer injection projects. In addition to the basic effects discussed so far, the viscosity and mobility-control of a polymer flood tends to be significantly affected by the fluid chemistry of the injected and resident water. More advanced models of polymer flooding therefore account for pH effects, salts, etc. Likewise, to achieve better oil recovery, polymer is often combined with other EOR processes, such as surfactant-polymer flooding, alkali-surfactant-polymer (ASP) flooding, polymer-alternating-gas (PAG) [8] processes, etc., within which polymer plays an important role for mobility ratio control.

Herein, we will introduce a simulator framework that has been developed on top of the open-source MATLAB Reservoir Simulation Toolbox [16] as a versatile and flexible test bench for rapid prototyping of new models of polymer flooding. The simulator is – like most commercial simulators – based on a black-oil formulation with simple first-order, upstream weighting for spatial discretization and fully implicit time stepping. This offers unconditional stability for a wide range of physical flow regimes and reservoir heterogeneities. Moreover, combining the fully implicit formulation with automatic differentiation ensures that it is simple to extend the basic flow models with new constitutive relationships, extra conservation equations, new functional dependencies, etc. By using numerical routines and vectorization from MATLAB combined with discrete differential and averaging operators from MRST, these equations can be implemented in a very compact form that is close to the mathematical formulation [6, 10]. Once you have implemented the discrete equations, the software will generate the discretizations and linearizations needed to obtain a working simulator that by default is designed to run on general unstructured grids. To test the performance of your new simulator, you can use one of the many grid factory routines and routines for generating petrophysical data to set up simplified and idealized test cases with a large variety of structured and unstructured grid formats in two and three spatial dimensions. Alternatively, you can also use the functionality for reading and parsing commercial input decks to set up proper validation on test cases having the complexity encountered in the daily work of reservoir engineers.

Using a scripting language like MATLAB will generally introduce a computational overhead, which can be quite significant for small systems. In our experience, however, the lack of computational efficiency is by far out-weighted by a more efficient development process, which is largely independent on your choice of operating system. At any point, you can stop the execution of the simulator to inspect your data, modify their values or the data structure itself, execute any number of statements and function calls, and go back and reiterate parts of the program, possibly with modified or additional data. For larger systems, the majority of the computational time of a well-designed simulator should be spent processing floating-point numbers. For this, MATLAB is efficient and fully comparable with compiled languages. Tests on two- and three-phase models with the order of ten to hundred thousand cells show that MRST simulators based on automatic differentiation are between two and ten times slower than fully optimized commercial simulators.

In the following, we will review the basic flow equations of polymer flooding and discuss how to formulate an efficient strategy that uses a separate inner Newton iteration process within the global nonlinear solution process. We then introduce key elements of the MRST software in some more detail and outline how we have applied the flexible grid structure, discrete differential operators, automatic differentiation, and object-oriented framework, to develop a new and efficient polymer

simulator that is readily applicable to simple conceptual models as well as models having the full complexity of real assets. We end the paper by presenting a series of numerical test cases for verifying and validating the simulator. To make new simulator prototypes capable of handling realistic flow models on large models, the underlying framework offers CPR-type preconditioners in combination with multigrid linear solvers, automated time-step selection, etc. In a recent paper, we also discussed how to formulate sequential solution strategies and use these to introduce a highly efficient multiscale pressure solver for polymer flooding [3].

2 Model Equations

In this section we will state our physical assumptions and outline the flow equations for polymer flooding, which are built as an extension of a general black-oil model.

2.1 The black-oil model

The black-oil model is a special multicomponent, multiphase flow model with no diffusion among the fluid components. The name 'black-oil' refers to the assumption that various hydrocarbon species can be lumped together to form two components at surface conditions – a heavy hydrocarbon component called 'oil' and a light component called 'gas'. At reservoir conditions, the two components can be partially or completely dissolved in each other, depending on the pressure, forming a liquid oleic phase and a gaseous phase. In addition, there is an aqueous phase, which herein is assumed to consist of only water. The corresponding continuity equations read,

$$\partial_t(\phi b_w s_w) + \nabla \cdot (b_w \mathbf{v}_w) - b_w q_w = 0, \tag{1a}$$

$$\partial_t[\phi(b_o s_o + b_g r_v s_g)] + \nabla \cdot (b_o \mathbf{v}_o + b_g r_v \mathbf{v}_g) - (b_o q_o + b_g r_v q_g) = 0, \tag{1b}$$

$$\partial_t[\phi(b_g s_g + b_o r_s s_o)] + \nabla \cdot (b_g \mathbf{v}_g + b_o r_s \mathbf{v}_o) - (b_g q_g + b_o r_s q_o) = 0. \tag{1c}$$

Here, ϕ is the porosity of the rock while s_α denotes saturation, p_α phase pressure, and q_α the volumetric source of phase α. The inverse formation-volume factors b_α, which measure the ratio between the bulk volumes of a fluid component occupied at surface and reservoir conditions, and the gas-oil ratio r_s and oil-gas ratio r_v, which measure the volumes of gas dissolved in the oleic phase and oil vaporized in the gaseous phase, respectively, are all user-specified functions of phase pressures. The phase fluxes $\mathbf{v}_\alpha$ are given from Darcy's law

$$\mathbf{v}_\alpha = -\lambda_\alpha \mathbf{K}(\nabla p_\alpha - \rho_\alpha g \nabla z), \qquad \alpha = o, w, g. \tag{2}$$

Here, $\mathbf{K}$ is the absolute permeability of the reservoir rock, while $\lambda_\alpha = k_{r\alpha}/\mu_\alpha$ is the mobility of phase α, where $k_{r\alpha}$ is the relative permeability and μ_α is the phase viscosity. The model is closed by assuming that the fluids fill the pore space completely, $s_o + s_w + s_g = 1$, and by supplying saturation-dependent capillary functions that relate the phase pressures. Altogether, the equation system will have three primary unknowns. Since we are going to study water-based EOR, we choose the first two to be water pressure p_w and water saturation s_w. The third unknown will depend on the phases present locally in each cell: If only the aqueous

and liquid phases are present, we choose r_s, whereas r_v is chosen when only the aqueous phase is present. If all phases are present, r_s and r_v depend on pressure and we hence choose s_g as the last unknown.

To get a complete model, we also need to support initial and boundary conditions. Herein, we will only consider problems with no-flow conditions on the outer boundaries and assume that the initial condition is supplied entirely by the user, e.g., as a hydrostatic pressure and fluid distribution. In addition, we need extra well equations to compute the volumetric source terms q_α. To this end, we use a semi-analytical model [21]

$$q_\alpha = \lambda_\alpha W_I(p_w - p), \tag{3}$$

where p is the reservoir pressure (inside a grid cell) and p_w is the pressure inside the wellbore. The well index W_I accounts for rock properties and geometric factors affecting the flow. The flow inside the wellbore is assumed to be instantaneous, so that fluids injected at the surface will enter the reservoir immediately. Likewise, the wellbore is assumed to be in hydrostatic equilibrium, so that the pressure at any point can be computed as a hydrostatic pressure drop from a datum point called the bottom hole, i.e., $p_w = p_{bh} + \Delta p_h(z)$. Wells are typically controlled by surface rate or the bottom-hole pressure. These controls are given as a set of extra equations that impose target values for fluid rates and bottom-hole pressures. And also, a certain logic that determines what happens if the computed rates or pressures violate operational constraints, in which case a well may switch from rate control to pressure control, shut in hydrocarbon rates become too low, etc.

2.2 The polymer model

In the present model, we assume that polymer is transported in the aqueous phase and that polymer changes the viscosity of this phase, but does not affect the liquid oleic and gaseous phases. The corresponding continuity equation reads,

$$\partial_t\big(\phi(1 - s_{ipv})b_w s_w c\big) + \partial_t\big(\rho_r c^a(1 - \phi)\big) + \nabla \cdot (b_w \mathbf{v}_p c) - b_w q_w c = 0. \tag{4}$$

Here, $c \in [0, c^*]$ is the polymer concentration given in units of mass per volume of water and c^* is the maximum possible concentration, $c^a = c^a(c)$ is the polymer adsorption concentration, ρ_r is the density of the reservoir rock, and s_{ipv} is the inaccessible (or dead) pore volume. The reduced mobility of the mixture of pure water and diluted polymer is modeled by introducing effective mixture viscosities $\mu_{w,\text{eff}}$ and $\mu_{p,\text{eff}}$ that depend upon the polymer concentration. This gives modified Darcy equations of the form,

$$\mathbf{v}_w = -\frac{k_{rw}(s_w)}{\mu_{w,\text{eff}}(c)R_k(c)}\,\mathbf{K}\big(\nabla p_w - \rho_w g \nabla z\big), \tag{5}$$

$$\mathbf{v}_p = -\frac{k_{rw}(s_w)}{\mu_{p,\text{eff}}(c)R_k(c)}\,\mathbf{K}\big(\nabla p_w - \rho_w g \nabla z\big). \tag{6}$$

Here, the non-decreasing function $R_k(c)$ models the reduced permeability experienced by the water–polymer mixture as a result of adsorption of polymer onto the rock's surface.

Inaccessible pore space. Many polymer flooding experiments show that polymer propagates faster through a porous medium than an inert chemical tracer dissolved in the polymer solution [13]. There are two reasons for this: First of all, large-sized polymer molecules cannot enter narrow pore throats and dead-end pore channels. Secondly, the free tumbling of polymer molecules is only possible at the center of the pore channels, away from the surface of the pore walls. Hence, the polymer solution can only flow through a fraction of the pore space. In (4), this effect is modeled by the scalar rock parameter s_{ipv}, which is defined as the amount of the pore volume inaccessible to the polymer solution for each specific rock type [22].

Adsorption. Polymer may attach to the rock surface through physical adsorption, which will reduce the polymer concentration and introduce a resistance to flow that reduces the effective permeability of water. This process is assumed to be instantaneous and reversible and is modeled through the accumulation term $\rho_r c^a(1-\phi)$ in (4).

Permeability reduction. The rock's effective permeability to water can be reduced, primarily by polymer adsorption but also as a result of polymer molecules that become lodged in narrow pore throats. The permeability reduction R_k representing this effect is given as,

$$R_k(c, c_{\max}) = 1 + (\mathrm{RRF} - 1)\frac{c^a(c, c_{\max})}{c^a_{\max}}, \qquad c_{\max}(x,t) = \max_{s \le t} c(x,s), \tag{7}$$

where $c^a_{\max}$ is the maximum adsorbed concentration and the hysteretic residual resistance factor $\mathrm{RRF} \geq 1$ is defined as the ratio between water permeability measured before and after polymer flooding. Both these quantities depend on the rock type.

Effective viscosities. To compute the effective viscosities of the water–polymer mixture, we will use the Todd–Longstaff mixing model [25]. In this model, the degree of mixing of polymer into water is represented by a mixing parameter $\omega \in [0,1]$, which generally depends on the displacement scenario, the geological heterogeneity, etc. If $\omega = 1$, water and polymer are fully mixed, whereas the polymer solution is completely segregated from pure water if $\omega = 0$. Let $\mu_{fm} = \mu_{fm}(c)$ denote the viscosity of a fully mixed polymer solution, then the effective polymer viscosity is calculated as

$$\mu_{p,\mathrm{eff}} = \mu_{fm}(c)^{\omega} \cdot \mu_p^{1-\omega}, \tag{8}$$

where $\mu_p = \mu_{fm}(c^*)$. The standard way of defining μ_{fm} is to write $\mu_{fm} = m_\mu(c)\mu_w$, where the viscosity multiplier m_μ is a user-prescribed function. The partially mixed water viscosity is calculated in a similar way as

$$\mu_{pm} = \mu_{fm}(c)^{\omega} \cdot \mu_w^{1-\omega}. \tag{9}$$

The effective water viscosity is then calculated by summing the contributions from the polymer solution and the pure water. Setting $\bar{c} = c/c^*$ results in the following alternative expression

$$\frac{1}{\mu_{w,\mathrm{eff}}} = \frac{1-\bar{c}}{\mu_{pm}} + \frac{\bar{c}}{\mu_{p,\mathrm{eff}}}, \qquad \mu_{w,\mathrm{eff}} = \frac{m_\mu(c)^{\omega}\,\mu_w}{1 - \bar{c} + \bar{c}/m_\mu(c^*)^{1-\omega}}. \tag{10}$$

2.3 Rheology of the polymer solution

The viscosity (or thickness) of a fluid is defined as the ratio between the shear stress and the shear rate and measures the resistance of a fluid mass to change its form. The aqueous, oleic, and gaseous phase in the black-oil model all have Newtonian viscosity, which means that the viscosity is independent of the experienced shear rate and can be modeled as a constant or as a pressure and/or temperature-dependent quantity. Polymer solutions, on the other hand, generally have shear-thinning viscosities. As shear rates increase, polymer molecules are elongated and aligned with the flow direction. Once this shear effect becomes sufficiently strong, the molecules will uncoil and untangle, causing a decrease in the effective viscosity of the water-polymer mixture. (Polymer solutions may also be shear-thickening, but this is less common).

Herein, we will represent shear effects using the same model as in a commercial simulator [22]. This model assumes that shear rate of water is proportional to the water velocity, as a result, the calculation of shear effect with this model is based on water velocity u_w. This assumption is not valid in general, but is reasonable when applied to a single rock type within reservoirs. A shear factor Z is introduced to describe the shear effect, which is calculated as

$$Z = \frac{\mu_{w,\mathrm{sh}}(u_{w,\mathrm{sh}})}{\mu_{w,\mathrm{eff}}} = \frac{1 + (m_\mu(c) - 1)m_{\mathrm{sh}}(u_{w,\mathrm{sh}})}{m_\mu(c)}, \tag{11}$$

where the multiplier $m_{\mathrm{sh}} \in [0, 1]$ is a user-prescribed function of the unknown shear-modified water velocity $u_{w,\mathrm{sh}}$, $\mu_{w,\mathrm{eff}}$ is the effective water viscosity (10) without considering the shear effect. With no shear effect ($m_{\mathrm{sh}} = 1$), we recover the effective water viscosity, whereas the shear viscosity equals $\mu_{w,\mathrm{eff}}/m_\mu(c)$ in the case of maximum shear thinning ($m_{\mathrm{sh}} = 0$). To calculate the unknown velocity $u_{w,\mathrm{sh}}$, we first introduce the effective water velocity $u_{w,0}$ computed from (5) with no shear effect, and then use the relation

$$u_{w,\mathrm{sh}} = u_{w,0}\frac{\mu_{w,\mathrm{eff}}}{\mu_{w,\mathrm{sh}}(u_{w,\mathrm{sh}})}$$

combined with (11) to derive the following implicit equation for $u_{w,\mathrm{sh}}$,

$$u_{w,\mathrm{sh}}\big[1 + (m_\mu(c) - 1)m_{\mathrm{sh}}(u_{w,\mathrm{sh}})\big] - m_\mu(c)u_{w,0} = 0. \tag{12}$$

Here, $u_{w,0}$ is the un-sheared water velocity.

Once (12) is solved for $u_{w,\mathrm{sh}}$, we can calculate shear factor Z from (11) and calculate the shear-modified viscosity $\mu_{w,\mathrm{sh}}$ and $\mu_{p,\mathrm{sh}}$ as

$$\mu_{w,\mathrm{sh}} = \mu_{w,\mathrm{eff}}Z \quad \mu_{p,\mathrm{sh}} = \mu_{p,\mathrm{eff}}Z.$$

In practice, we compute the modified phase fluxes directly as

$$\mathbf{v}_{w,\mathrm{sh}} = \frac{\mathbf{v}_w}{Z} \qquad \mathbf{v}_{p,\mathrm{sh}} = \frac{\mathbf{v}_p}{Z}$$

to avoid repeated computation.

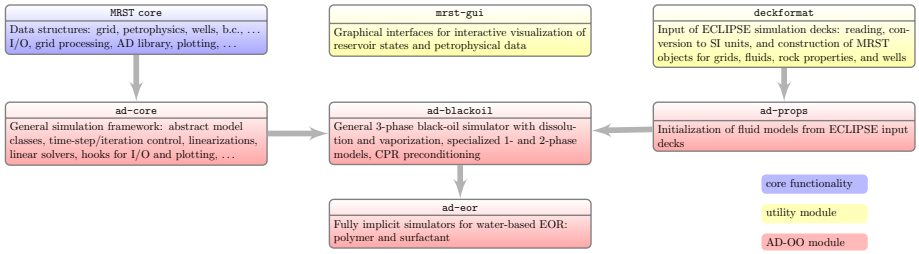


Fig. 1: Modules from MRST used to implement a fully implicit polymer simulator.

3 The Three-Phase Black-Oil Simulator in MRST

In this section, we will discuss how to discretize and solve the basic black-oil equations and how to implement these discretizations and solvers using functionality for rapid prototyping from the open-source MRST software to obtain a simulator framework that is efficient and simple to extend with new functionality. However, before we start discussing the discretizations and solvers, we give a brief introduction to [16]; more details can be found in [11], [6], and [10].

The essence of MRST is a relatively slim core module `mrst-core` that contains a flexible grid structure and a number of grid factory routines; routines for visualizing grids and data represented on cells and cell faces; basic functionality for representing petrophysical properties, boundary conditions, source terms and wells; computation of transmissibilities and data structures holding the primary unknowns; basic functionality for automatic differentiation (AD); and various low-level utility routines. The second, and by far the largest part of the software, is a set of add-on modules that implement discretizations and solvers; more complex data structures, extended grid formats, and visualization routines; more advanced AD functionality for building simulators; reading and processing of industry-standard input decks; as well as a wide variety of simulators, graphical user interfaces, and workflow tools. Many of these modules offer standalone functionality built on top of `mrst-core` and standard MATLAB routines. More advanced simulators and workflow tools, on the other hand, typically rely on functionality from many of the other MRST modules. The majority of the software that is publicly available is quite mature and well documented in a format similar to that used in standard MATLAB functions. Most modules also offer examples and tutorials written in a workbook format using cell-mode scripts. Herein, we focus on the AD-OO family of modules rapid prototyping of fully implicit simulators, see Figure 1.

3.1 Grids and discrete differentiation operators

When working with grids that are more complex than simple rectilinear boxes, one needs to introduce some kind of data structure to represent the geometry and topology of the grid. In MRST, we have chosen to use a quite rich format for unstructured grids, in which general geometric and topological information is always present and represented explicitly regardless of whether a specific grid allows for simplifications. The reason for this is that we want to ensure interoperability among different grid types and computational tools, and ensure maximal flexibility

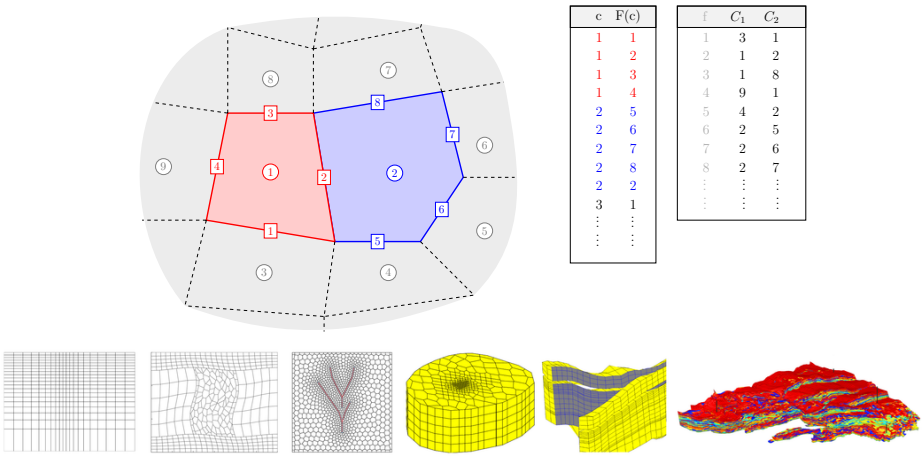


Fig. 2: Illustration of grids in MRST. The upper plot shows the relation between cells and faces which can be used to define discrete differentiation operators. The lower plots show various grids of increasing complexity, from a simple rectilinear grid to a model of the Gullfaks field from the Norwegian North Sea.

when developing new methods. As a result, grid and petrophysical properties are passed as input to almost all simulation and workflow tools in MRST. For standard low-order, finite-volume discretizations one does not need all this information and in many simulators, grid and petrophysical parameters are only seen explicitly by the preprocessor, which constructs a connection graph with cell properties and pore volumes associated with vertices and inter-cell transmissibilities associated with edges. To simplify the presentation, we will herein only discuss this information and show how it can be used to build abstract operators implementing powerful averaging and discrete differential operators that later will enable us to write the discrete flow equations in a very compact form.

Figure 2 illustrates parts of the unstructured grid format, in which grids are assumed to consist of a set of matching polygonal (2D) or polyhedral (3D) cells with matching faces. These grids are represented using three data objects – cells, faces, and nodes – which contain the geometry and topology of the grid. To form our discrete differential operators, we basically need two mappings. The first is the map $F(c)$, which for each cell gives the faces that bound the cell. The second is a mapping that brings you from a given cell face f to the two cells $C_1(f)$ and $C_2(f)$ that lie on opposite sides of the face. In the following we will use boldfaced letters to represent arrays of discrete quantities and use the notation $\mathbf{q}[c]$ and $\mathbf{q}[f]$ to denote the element of an array $\mathbf{q}$ corresponding to grid cell c and cell face f, respectively.

We can now define discrete counterparts of the continuous divergence and gradient operators. The `div` operator is a linear mapping from faces to cells. If $\mathbf{v}[f]$ denotes a discrete flux over face f with orientation from cell $C_1(f)$ to cell $C_2(f)$, then the divergence of this flux restricted to cell c is given as

$$\texttt{div}(\mathbf{v})[c] = \sum_{f \in F(c)} \operatorname{sgn}(f)\mathbf{v}[f], \qquad \operatorname{sgn}(f) = \begin{cases} 1, & \text{if } c = C_1(f), \\ -1, & \text{if } c = C_2(f). \end{cases} \tag{13}$$

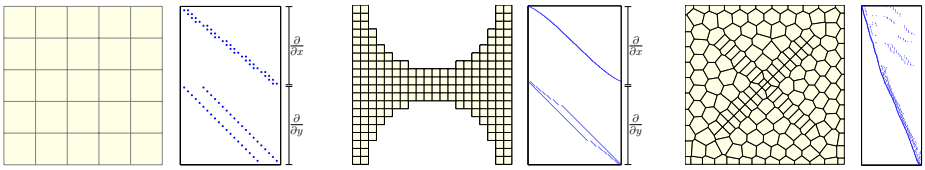


Fig. 3: The sparsity structure of the matrix $\mathbf{D}$ used to define discrete differential operators for three different 2D grids. The two Cartesian grids consist of two blocks that each have a clear banded structure.

The `grad` operator maps from cell pairs to faces. If, for instance, $\mathbf{p}$ denotes the array of discrete cell pressures, the gradient of this cell pressure restricted to face f is defined as

$$\texttt{grad}(\mathbf{p})[f] = \mathbf{p}[C_2(f)] - \mathbf{p}[C_1(f)]. \tag{14}$$

If we assume no-flow conditions on the outer faces, the discrete gradient operator is the adjoint of the divergence operator as in the continuous case, i.e.,

$$\sum_c \texttt{div}(\mathbf{v})[c]\,\mathbf{p}[c] + \sum_f \texttt{grad}(\mathbf{p})[f]\,\mathbf{v}[f] = 0.$$

Since `div` and `grad` are linear operators, they can be represented by a sparse matrix $\mathbf{D}$ so that $\texttt{grad}(\mathbf{x}) = \mathbf{D}\mathbf{x}$ and $\texttt{div}(\mathbf{x}) = -\mathbf{D}^T\mathbf{x}$. Figure 3 shows the sparsity structure of $\mathbf{D}$ for three different 2D grids. In addition, we need to define the *transmissibilities* that describe the flow across a cell face f given a unit pressure drop between the two neighboring cells $i = C_1(f)$ and $k = C_2(f)$. To this end, let $A_{i,k}$ denote the area of the face, $\mathbf{n}_{i,k}$ the normal to this face, and $\mathbf{c}_{i,k}$ the vector from the centroid of cell i to the centroid of the face. Then, the transmissibility is defined as

$$\mathbf{T}[f] = \left[T_{i,k}^{-1} + T_{k,i}^{-1}\right]^{-1}, \qquad T_{i,k} = A_{i,k}\mathbf{K}_i \frac{\mathbf{c}_{i,k}\cdot\mathbf{n}_{i,k}}{|\mathbf{c}_{i,k}|^2}, \tag{15}$$

where $\mathbf{K}_i$ is the permeability tensor in cell i with primal axes aligned with the grid axes. To provide a complete discretization, we also need to supply averaging operators that can map rock and fluid properties from cells to faces. For this, we will mainly use arithmetic averaging, which in its simplest form can be written

$$\texttt{avg}_a(\mathbf{q})[f] = \tfrac{1}{2}\left(\mathbf{q}[C_1(f)] + \mathbf{q}[C_2(f)]\right).$$

3.2 Discrete flow equations for black-oil

The discrete operators defined above can be used to discretize the flow equations in a very compact form. If we use a first-order, implicit temporal discretization and a standard two-point spatial discretization with upstream weighting, the discrete conservation for the aqueous phase can be written as

$$\begin{aligned}\frac{1}{\Delta t}\Big(\phi[c]\,\mathbf{b}[c]\,\mathbf{s}[c]\Big)^{n+1} - \frac{1}{\Delta t}\Big(\phi[c]\,\mathbf{b}[c]\,\mathbf{s}[c]\Big)^{n} \\ +\texttt{div}(\mathbf{b}\mathbf{v})[c]^{n+1} - \big(\mathbf{b}[c]\mathbf{q}[c]\big)^{n+1} = 0,\end{aligned} \tag{16}$$

where we have omitted the subscript 'w' for brevity. To evaluate the product of the inverse formation-volume factor and the phase flux at the cell interfaces, we introduce the operator for extracting the upstream value

$$\mathtt{upw}(\mathbf{h})[f] = \begin{cases} \mathbf{h}[C_1(f)], & \text{if } \mathtt{grad}(\mathbf{p})[f] - g\,\mathtt{avg}_a(\rho)[f]\mathtt{grad}(\mathbf{z})[f] > 0, \\ \mathbf{h}[C_2(f)], & \text{otherwise.} \end{cases} \tag{17}$$

Then, the discrete version of Darcy's law multiplied by b_w reads

$$(\mathbf{bv})[f] = -\mathtt{upw}(\mathbf{b}\lambda)[f]\,\mathbf{T}[f]\,\Big(\mathtt{grad}(\mathbf{p})[f] - g\,\mathtt{avg}_a(\rho)[f]\,\mathtt{grad}(\mathbf{z})[f]\Big). \tag{18}$$

If we now collect the discrete conservation equations for the aqueous, oleic, and gaseous phases along with the well equations – all written on residual form – we can write the resulting system of nonlinear equation as

$$\mathbf{R}(\mathbf{x}) = \mathbf{0}, \tag{19}$$

where $\mathbf{x}$ is the vector of unknown state variables at the next time step. The standard way to solve such a nonlinear system is to use Newton's method. That is, we write $\mathbf{x} = \mathbf{x}^0 + \Delta\mathbf{x}$, and use a standard multidimensional Taylor expansion to derive the iterative scheme,

$$\mathbf{J}(\mathbf{x}^i)\big(\mathbf{x}^{i+1} - \mathbf{x}^i\big) = -\mathbf{R}(\mathbf{x}^i), \tag{20}$$

where $\mathbf{J} = d\mathbf{R}/d\mathbf{x}$ is the Jacobian matrix of the residual equations.

3.3 Automatic differentiation in MRST

Before continuing to describe our implementation of the black-oil simulator, we give a quick introduction to automatic differentiation (AD) for the benefit of readers not familiar with this powerful technique. Automatic differentiation – also known as algorithmic or computational differentiation – is a set of techniques for simultaneous numerical evaluation of a function and its derivatives with respect to a set of predefined primary variables. The key idea of AD is that every function evaluation will execute a sequence of elementary arithmetic operations and functions, for which analytical derivatives are known. To exemplify, let x be a scalar variable and $f = f(x)$ an elementary function. The AD representations are $\langle x, 1\rangle$ and $\langle f, f_x\rangle$, where 1 is the derivative dx/dx and f_x is the numerical value of the derivative $f'(x)$. By applying the chain rule in combination with standard rules for addition, subtraction, multiplication, division, and so on, we can now automatically compute derivatives to within machine precision, e.g., addition: $\langle f, f_x\rangle + \langle g, g_x\rangle = \langle f + g, f_x + g_x\rangle$, cosine: $\cos(\langle f, fx\rangle) = \langle \cos(f), -\sin(f)f_x\rangle$, etc. The same principle can easily be extended to higher-order derivatives and partial derivatives of functions of multiple variables.

In MATLAB, this functionality can be elegantly implemented using classes and operator overloading. When MATLAB encounters an expression `a+b`, the software will choose one out of several different addition functions depending on the data types of `a` and `b`. All we now have to do is introduce new addition functions for the various classes of data types that `a` and `b` may belong to. You can read more about how this is done in [17]. MRST implements automatic differentiation as part of

`mrst-core`. A new AD variable is instantiated by the call `x=initVariablesAD(x0)`, where `x0` is a scalar or an array containing values to be used for subsequent function evaluations. Any new variable `f` computed based on `x` will now automatically become an AD variable, whose value and derivatives are accessed as `f.val` and `f.jac`, respectively. The AD class in `mrst-core` differs from most other libraries in the sense that instead of representing the Jacobian with respect to multiple variables as a single matrix, we have chosen to let `jac` be a list of sparse matrices that each represents the derivatives with respect to a single primary variable. In solution algorithms, one may want to separate pressure, compositions, and variables associated to wells, and by keeping the sub-Jacobians separate and not assembling directly into one large sparse matrix, we avoid manipulating subsets of large sparse matrices, which has low performance in MATLAB. Moreover, this approach makes it simpler for users who wish to manipulate matrix blocks that represent specific sub-equations in the Jacobian of a full equation system.

3.4 Making a black-oil simulator: procedural approach

Having introduced you to automatic differentiation, we will now show how this idea can be used to implement a fully implicit solver for the discrete black-oil equations on residual form (16). To keep track of all the different entities that are part of the simulation model, MRST introduces a number of data objects:

- a state object, which basically is a MATLAB structure holding arrays with the unknown pressures, saturations, concentrations, and inter-cell fluxes, as well as unknowns associated with the wells;
- a grid structure `G`, which in particular implements the mappings F, C_1, and C_2 introduced in Section 3.1;
- a structure `rock` representing the petrophysical data, primarily porosity and permeabiliy, but also net-to-gross, multipliers that limit (or increase) the flow between neighoring cells, etc;
- a structure `fluid` representing the fluid model, which is implemented as a collection of function handles that can be queried to give fluid densities and viscosities, evaluate relative permeabilities, formation volume factors, etc;
- additional structures that contain the global drive mechanisms, including wells and boundary conditions.

By convention, we collect `G`, `rock`, and `fluid` in an additional data structure called `model`, which also implements utility functions to access model behavior. Given a state object, we can for instance query values for physical variables

```
[p0, sW, sG, rs, rv, wellSol] = model.getProps(state, 'pressure', 'water', 'gas', ...
                                                'rs', 'rv', 'wellSol');
bhp  = vertcat(wellSol.bhp);
qWs  = vertcat(wellSol.qWs);          % ... and similarly for qOs and qGs
```

Here, the array `p0` holds one oil pressure value per cell, `sW` holds one water saturation value, etc. The last output, `wellSol`, contains a list of data structures, one for each well, that contain the unknowns associated with the perforations of each well. The call to `vertcat` collects these quantities into standard arrays. Which among s_g, r_s, and r_v one should choose as primary reservoir variable will vary from one cell to

the other depending the fluid phases present. For brevity, we assume that all three phases are always present, so that r_s and r_v are functions of pressure. Thus, we henceforth use `sG` as our third unknown. Having extracted all the primary variables needed, we set them to be AD objects

```
[pO, sW, sG, qWs, qOs, qGs, bhp] = initVariablesADI(pO, sW, sG, qWs, qOs, qGs, bhp);
```

When these AD objects are used to evaluate fluid and rock-fluid properties in the cells, we will also get derivatives with respect to the primary variables evaluated at their current value

```
[krW, krO, krG] = model.evaluateRelPerm({sW, 1- sW - sG, sG}); % relative permeabilities
bW    = model.fluid.bW(p);                                     % inverse formation-volume factor
rhoW  = bw .* model.fluid.rhoWS;                               % density at reservoir conditions
mobW  = krW ./ model.fluid.muW(p);                             % mobility
```

To evaluate Darcy's law across each face, we need to use the averaging operator, the gradient, and the transmissibility introduced in Section 3.1. In MRST, the corresponding mappings are computed during the preprocessing phase based on `G` and `rock` and stored in terms of function handles in structure `operators` inside the `model` objects. For brevity, we henceforth refer to this as `ops`. We first use the averaging operator and the gradient operator to pick the upstream directions for each interface

```
rhoWf = ops.faceAvg(rhoW);                  % density at cell faces
gdz   = model.getGravityGradient();         % g*nabla(z)
pW    = pO - model.fluid.pcOW(sW);          % water pressure
dpW   = ops.Grad( pW ) - rhoWf.*gdz;        % derivative terms in Darcy's law
upcw  = ( double(dpW) <= 0);                % upwind directions
```

Then, we use the upstream operator (17), which is also contained in `ops`, to compute the correct water fluxes for all interior interfaces

```
bWvW  = ops.faceUpstr(upcw, bW.*mobW).*ops.T.*dpW;
```

The last thing we need to do is to handle the pressure-dependence of the accumulation term. In MRST, this is represented as a static pore volume, evaluated at a reference pressure, and a pressure-dependent multiplier function

```
[pvMult, pvMult0] = getMultipliers(model.fluid, pO, pO0);
```

Having computed all the necessary values in cells and on faces, we can evaluate the residual from the homogeneous part of the aqueous conservation equation over a time step `dt`

```
water = (ops.pv/dt).*( pvMult.*bW.*sW - pvMult0.*bW0.*sW0 ) + ops.Div(bWvW);
```

The homogeneous residual equations for the oleic and gaseous phases are computed in the same way, and then we collect the three phase equations in a cell array holding all reservoir equations:

```
eqs = {water, oil, gas};
```

To form a complete model, we also need to add residual equations for wells and incorporate the effects of driving forces into the continuity equations. Computing the contributions from wells, volumetric source terms, and boundary conditions

is a bit more involved and skipped for brevity. However, once the resulting fluxes or source terms have been computed, all we need to do is subtract them from the continuity equations in the affected cells. Looking at the overall implementation, it is clear that there is an almost one-to-one correspondence between continuous and discrete variables. In particular, the code implementing the conservation equations is almost on the same form as (16), except that compressibility has been included through a pressure-dependent multiplier instead of evaluating a function $\phi(p)$ directly. Likewise, you may notice the absence of indices and that there are no loops running over cells and faces. Using discrete averaging and differential operators derived from a general unstructured grid format means that the discrete equations can be implemented once and for all without knowing the specifics of the grid or the petrophysical parameters. This is a major advantage that will greatly simplify the process of moving from simple Cartesian cases to realistic geological models.

With the code above, we have collected the whole model into a cell array that contains seven different residual equations (three continuity equations, three equations for well rates, and one equation providing well control) as well as their Jacobian matrices with respect to the primary variables (p_o, s_w, s_g, p_{bh} and q_α^s). The last thing we need to do to compute one Newton iteration is to assemble the global Jacobian matrix and compute the Newton update (20).

```
eq = cat(eqs{:});                % concatenate unknowns and assemble Jacobian
J  = eq.jac{1};                  % extract Jacobian
upd = -(J \ eq.val);             % compute Newton update for all variables
```

This shows the strength of using automatic differentiation. There is no need to compute linearizations explicitly; these are computed implicitly by operator overloading when we evaluate each residual equation. Likewise, we do not need to explicitly assemble the overall Jacobian matrix; this is done by MATLAB and MRST when we concatenate the cell array of AD variables. All that remains to get a first prototype solver is to specify two loops, an outer loop that advances the time steps, and an inner loop that keeps computing Newton updates until the residual is sufficiently small, as shown to the left in Figure 4. The result is a framework that is very simple to extend with new functionality [6,10]: you can implement new fluid behavior or add extra conservation equations, and the AD functionality will automatically generate the correct linearized equations. However, to get a simulator capable of running industry-grade simulations, we will need to introduce more sophisticated numerical methods.

3.5 Object-oriented implementation in MRST

The actual code lines presented above are excerpts of `equationsBlackOil` from the `ad-blackoil` module, and have been slightly modified for pedagogical purposes. Industry-standard reservoir models contain many details that are seldom discussed in scientific papers. For brevity, we skipped a lot of details like conversions and consistency checks, and did not include various multipliers used to manipulate the flow between neighboring cells. Likewise, we did not discuss construction of reverse flow equations that can be used to compute adjoints [4]. However, since MRST is open-source, the interested reader can consult the code for full details.

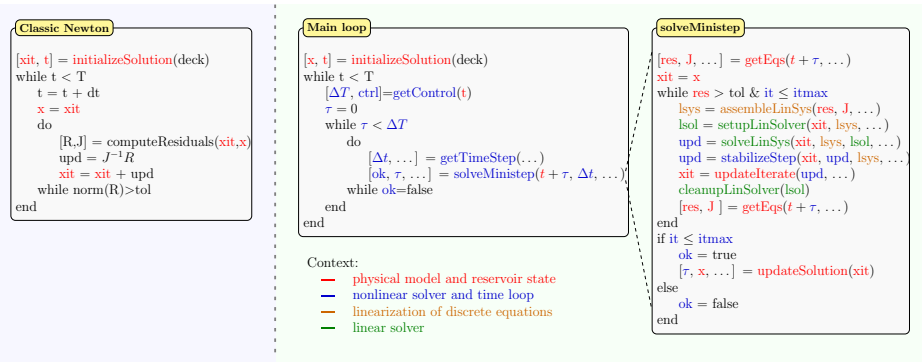


Fig. 4: Comparison of a simple time-loop with a standard Newton solver and the more sophisticated approach used in the `ad-core` framework, in which the time-loop has been organized into specific numerical contexts to separate the implementation of physical models, discrete equations and linearizations, nonlinear solvers and time-step control, and linear solvers. Here, `solveMinistep` subdivides well-control intervals into smaller time steps, specified by user or error control.

In all their generality, black-oil models can be very computationally challenging for a number of reasons: the flow equations have a mixed elliptic-hyperbolic character; there are order-of-magnitude variations in parameters and spatial and temporal constants; primary variables can be strongly coupled through various (delicate) force balances that shift throughout the simulation; fluid properties can have discontinuous derivatives and discontinuous spatial dependence; and grids representing real geology will have cells with rough geometries, large aspect ratios, unstructured connections through small face areas, etc. As a result, the simple Newton strategy discussed above will unfortunately not work very well in practice. Linearized flow problems typically have very large condition numbers, and while we can rely on the direct solvers in MATLAB being efficient for small systems, iterative solvers are needed for larger systems. These will not converge efficiently unless we also use efficient preconditioners that account for strong media contrasts and the mixed elliptic-hyperbolic character of the flow equations. To ensure that saturations stay within their physical bounds, each Newton update needs to be accompanied by a stabilization method that either crops, dampens, or performs a line search along the update directions. Likewise, additional logic is needed to map the updated primary variables back to a consistent reservoir state, switch primary variables as phases appear or disappear, trace changes in fluid components to model hysteretic behavior, etc. To get a robust simulator, we also need to introduce sophisticated time-step control that monitors the iteration and cuts the time step if this is deemed necessary to improve convergence. And finally, we need procedures for updating the well controls in response to changes in the reservoir state and the injection and production of fluids.

Introducing all this functionality in a procedural code is possible, but can easily give unwieldy code. A lot of this functionality is also to a large degree generic and can be reused from one model/simulator to another. One way to design a transparent and well-organized code is to divide the simulation loop into different numerical

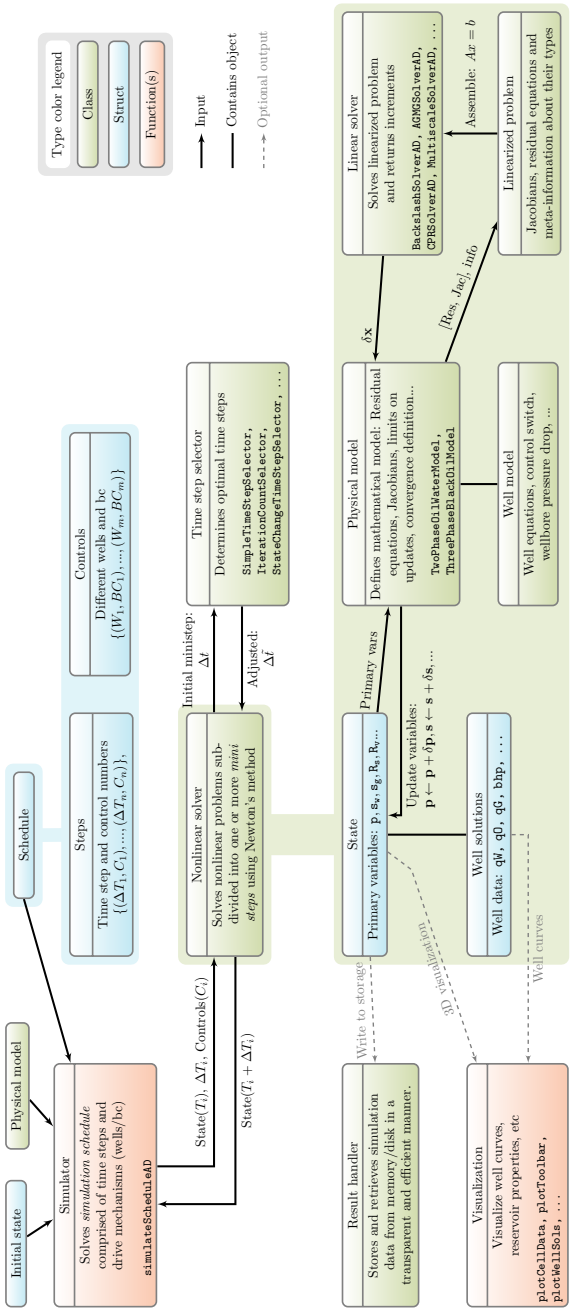


Fig. 5: Overview of how components in the object-oriented AD framework are organized to implement a black-oil simulator. The different components are colorized by the type of the corresponding construct (class, struct, or function). Notice, in particular, how the nonlinear solver utilizes multiple components to solve each ministep on behalf of the simulator function.

contexts, e.g., as outlined in Figure 4, and only expose the details that are needed within each of these contexts. This motivated us to develop the `ad-core` module (see [6]), which offers an object-oriented AD framework that enables the user to separate physical models and reservoir states, nonlinear solvers and time loops, discrete flow equations and linearizations, and linear solvers. The framework has been tailor-made to support rapid prototyping of new reservoir simulators based on fully implicit or sequentially implicit formulations and contains a lot of functionality that is specific for reservoir simulators. Figure 5 outlines how various classes, structures, and functions can be organized to formulate an efficient black-oil simulator. In particular, the time-step selectors implement simple heuristic algorithms like the Appleyard and modified Appleyard chop as used in commercial simulators. There are linear-solver classes that implement a state-of-the-art, constrained pressure residual (CPR) preconditioner [2], which can be combined with efficient algebraic multigrid solvers like the aggregation-based method of [19]. Notice also that assembly of the linearized system is relegated to a special class that stores meta-information about the residual equations (i.e., whether they are reservoir, well, or control equation) and the primary variables. This information is useful when setting up preconditioning strategies that utilize structures in the problem.

4 The Polymer Flooding Simulator

In this section, we will discuss how we can utilize the general framework presented above to implement a new polymer simulator capable of simulating real EOR scenarios. As in the previous section, we will focus on the main steps in the implementation and leave out a number of details that can easily be found by consulting the corresponding code from the `ad-eor` module (first released in MRST 2016a).

4.1 Defining the polymer model object

The first thing we need to do is to set up a physical model. Obviously, the black-oil model already has most of the features we need for our polymer simulator. To avoid duplicating code, this model has been implemented as the extension of a general model skeleton that specifies typical entities and features seen in reservoir models, and the skeleton model is in turn a special case of a generic physical model; see Figure 6. We use inheritance to leverage all this existing functionality:

```
classdef ThreePhaseBlackOilPolymerModel < ThreePhaseBlackOilModel
    properties
        polymer
        usingShear
    end
    methods
      :
```

The properties `polymer` and `usingShear` are boolean variables that tell whether polymer and shear effects are present or not. (For the moment, the first will be true and the second false). The next thing we need to do is to add two new variables: the concentration c, which will be a primary variable, and the secondary variable $c_{\max}$ holding the maximum observed polymer concentration, which will be needed to model the hysteretic behavior of the permeability reduction factor (7).

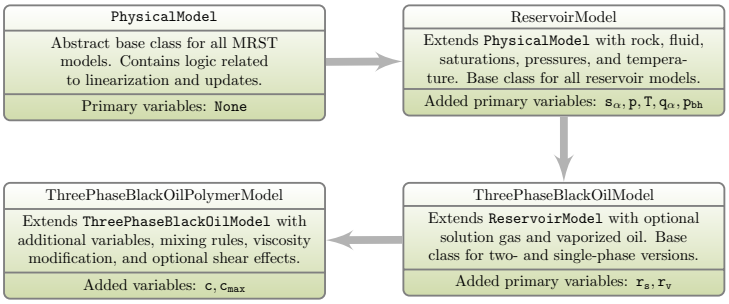


Fig. 6: The class hierarchy used to define the three-phase, black-oil, polymer model.

```
function [fn, index] = getVariableField(model, name)
  switch(lower(name))
    case {'polymer'}
      index = 1; fn = 'c';
    case {'polymermax'}
      index = 1; fn = 'cmax';
    otherwise
      [fn, index] = getVariableField@ThreePhaseBlackOilModel(model, name);
  end
```

Here, `index` tells the size of the variables in each cell, and the second last line provides access to all variables defined in the black-oil model. We also need to add one extra variable giving the surface rate of polymer in and out of wells. This is done in the constructor of the polymer object. The general framework also offers two additional member functions that provide useful hooks into the general simulation loop. The first function is run after every iteration update to enable us to check if the computed states are consistent with the underlying physics. We set this function to inherit all consistency checks from the black-oil model and additionally enforce that $c \in [0, c^*]$.

```
function [state, report] = updateState(model, state, problem, dx, drivingForces)
  [state, report] = updateState@ThreePhaseBlackOilModel(model,...
    state, problem, dx, drivingForces);
  if model.polymer
    c = model.getProp(state, 'polymer');
    c = min(c, model.fluid.cmax);
    state = model.setProp(state, 'polymer', max(c, 0));
  end
```

The second function is run after the nonlinear equation has converged and can be used, e.g., to model hysteretic behavior as in our R_k function.

```
function [state, report] = updateAfterConvergence(model, state0, state, ...
                                                  dt, drivingForces)
  [state, report] = updateAfterConvergence@ThreePhaseBlackOilModel(model, ...
    state0, state, dt, drivingForces);
  if model.polymer
    c     = model.getProp(state, 'polymer');
    cmax  = model.getProp(state, 'polymermax');
    state = model.setProp(state, 'polymermax', max(cmax, c));
  end
```

To form a complete model, we also need to incorporate functions and parameters describing the adsorption, the Todd–Longstaff mixing, inaccessible pore space, etc. Instead of implementing analytical formula (or hard-coded tables), we have chosen to get all necessary data by parsing industry-standard input decks. This parsing is automated in MRST in the sense that if you want a keyword `KWD` to be interpreted, you will have to implement a new function called `assignKWD` in the `ad-props` module. This function should take three parameters: the fluid object to which the property is appended, data values, and a structure containing region identifiers, which could possibly be used to incorporate spatial dependence in the parameters. For the six keywords describing our polymer model [22], implementing these functions amounted to approximately forty extra lines of code to interpret each keyword and setup functions that either extract constants or interpolate the tabulated data in the input deck correctly.

4.2 Discretized equations without shear effects

The last thing we have to do is to implement the discretized flow equations. That is, we need to implement the member function `getEquations` which the AD-OO framework will call whenever it needs to evaluate the residual of the flow equations

```
function [problem, state] = ...
  getEquations(model, state0, state, dt, drivingForces, varargin)
    [problem, state] = equationsThreePhaseBlackOilPolymer(state0, state, ...
        model, dt, drivingForces, varargin{:});
end
```

To this end, we start by copying and renaming the function `equationsBlackOil`, which was discussed above and implements the discretized equations for the three-phase black-oil model. The first change we need to introduce in the copied function is in the extraction of physical variables.

```
  [p0, sW, sG, rs, rv, c, cmax, wellSol] = model.getProps(state, ...
        'pressure', 'water', 'gas', 'rs', 'rv', 'polymer', 'polymermax', 'wellsol');
  :
  qWPoly = vertcat(wellSol.qWPoly);
```

Similar changes are also made when choosing and instantiating the primary variables as AD objects. In the computation of fluid properties, we start with the polymer properties since they will also affect the mobility of water.

```
  ads = fluid.ads(max(c, cmax));                    % adsorption term

  mixpar = fluid.mixPar;                            % mixing parameter w
  cbar   = c/fluid.cmax;                            % normalized concentration
  a = fluid.muWMult(fluid.cmax).^(1-mixpar);        % viscosity multiplier resulting
  b = 1./(1-cbar+cbar./a);                          % .. from mixing of water and
  muWeffMult = b.*fluid.muWMult(c).^mixpar;         % .. polymer using w-mixing rules

  permRed = 1 + ((fluid.rrf-1)./fluid.adsMax).*ads; % permeability reduction
  muWMult = muWeffMult.*permRed;                    % full multiplier for mobility
```

The computation of the water properties is almost as before, except for a minor change marked in red:

```
mobW = krW ./ ( fluid.muW(p) .*muWMult);
```

Apart from this, the computation of the aqueous, oleic, and gaseous residual remains unchanged. The residual equation for polymer is implemented as follows,

```
poro = ops.pv./G.cells.volumes;
polymer = (ops.pv.*(1−fluid.dps)/dt).*(pvMult.*bW.*sW.*c − ...
          pvMult0.*fluid.bW(p0).*sW0.*c0) + (ops.pv/dt).* ...
         (fluid.rhoR.*((1−poro)./poro).*(ads − ads0)) + ops.Div(bWvP);
```

which again is almost the same as the corresponding expression for the continuous equations. Unfortunately, using this residual equation without modifications may incur numerical unstabilities in the case when water is almost nonexistent. To prevent this, we detect all cells in which the diagonal element of the Jacobian falls below a certain lower tolerance and replace the residual in these cells by the polymer concentration, i.e., if `bad` is a logical array indexing these cells, we set `polymer(bad)=c(bad)`. Assuming that we make the necessary modifications to the well models, we now have four continuity equations, four well equations, and a control equation that can be linearized and assembled as before by first building a cell array of residual equations, which we then use to construct a `LinearizedProblem` object.

4.3 Including shear effects

The simulator, as described above, computes fluxes across cell faces and fluxes in and out of wells. However, to compute shear effects, we need the un-sheared water velocities, which can be defined on each face f of the discrete grid as

$$\mathbf{u}_{w,0}[f] = \frac{\mathbf{v}_w[f]}{\mathtt{avg}_a(\phi)[f]\,\mathbf{A}[f]},$$

where $\mathbf{A}[f]$ is the face area. The product ϕA is then the available area for the fluids to flow through each particular face. In addition, we need to evaluate the viscosity multiplier $m_\mu(c)$ at each face. This is done by picking the upstream value. Then we can instantiate $\mathbf{u}_{w,sh}$ as an AD variable, initialized by $\mathbf{u}_{w,0}$, and use a standard Newton iteration to solve (12). This inner iteration is invoked every time we need to evaluate the water or polymer residual.

The solution process for the shear factor calculation is presented in Figure 7. There are basically three parts involved. At the beginning, we initialize the AD variable `Vsh` based on the un-sheared velocity `Vw`, and set up the residual equation we are solving with function `shFunc`, which can be easily related to (12). Then a standard Newton iteration process is used to solve the residual equation. Finally, we calculate and return the shear factor `z` based on the calculated sheared water velocity `Vsh` following (11). The function solve for the shear factors of all the faces at the same time for better efficiency.

Shear effects are most important near the wellbore, and will to a large extent determine the injectivity of a polymer solution. Unfortunately, it is challenging to compute a representative shear rate. Grid cells are typically large compared with the length scale of the near-well flow, and using a simple average over the grid cell will tend to smear flow rates and hence underestimate the non-Newtonian

Shear Calculation

```
function z = computeShearMult(fluid, Vw, muWMultf)
    Vsh = Vw;      % give the initial guess for the Vsh
    Vsh = initVariablesADI(Vsh); % initialize the AD variable
    plyshearMult = fluid.plyshearMult; % get the shear function M

    shFunc = @(x) x.*(1+(muWMultf-1.).*plyshearMult(x))-muWMultf.*Vw; % residual function
    eqs = shFunc(Vsh);

    resnorm = norm(double(eqs), 'inf'); % initial norm of residual
    iter = 0;
    maxit = 30;  % maximum iteration number
    abstol = 1.e-15; % tolerance for convergence

    while (resnorm > abstol) && (iter <= maxit) % Newton iteration
      J = eqs.jac{1};  % Jacobian
      dVsh = -(J \ eqs.val); % Newton incremental update
      Vsh.val = Vsh.val + dVsh; % update the solution
      eqs = shFunc(Vsh);
      resnorm = norm(double(eqs), 'inf');  % norm of the residual
      iter = iter + 1;
    end

    if (iter >= maxit) && (resnorm > abstol) % not converged
        error('Convergence failure within %d iterations\nFinal residual = %.8e', maxit, resnorm);
    end

    if(resnorm <= abstol) % convergence achieved
        M = plyshearMult(Vsh.val);
        z = (1 + (muWMultf - 1.).* M) ./ muWMultf; % shear factor
    end
end
```

Fig. 7: Shear factor calculation with MRST

effects. One obvious remedy is to use local grid refinement (LGR) around the wells, see [14]. Another alternative is to use an analytical injectivity model in which the water velocity is computed at a representative radius from each well perforation [9, 22]. The representative radius is defined as $r_r = \sqrt{r_w\, r_a}$, where r_w is the wellbore radius and r_a is the areal equivalent radius of the grid cell in which the well is completed. The water velocity can then be computed as

$$u_{w,0} = \frac{q_w^s}{2\pi r_r h_{wc}\, \phi\, b_w},$$

where q_w^s is the surface water rate and h_{wc} is the height (or length) of the perforation inside the completed grid cell. Notice, however, that this model has only been derived assuming Cartesian cell geometries.

4.4 Running the simulator from an input deck

MRST is primarily a tool for prototyping new computational methods and simulator tools. As such, the software does not offer any simulator that can be called directly from the command line in MATLAB. Instead, the idea is that users should write the simulator scripts themselves, using functionality from the toolbox. The tutorials and module examples contain many such simulator scripts that can be

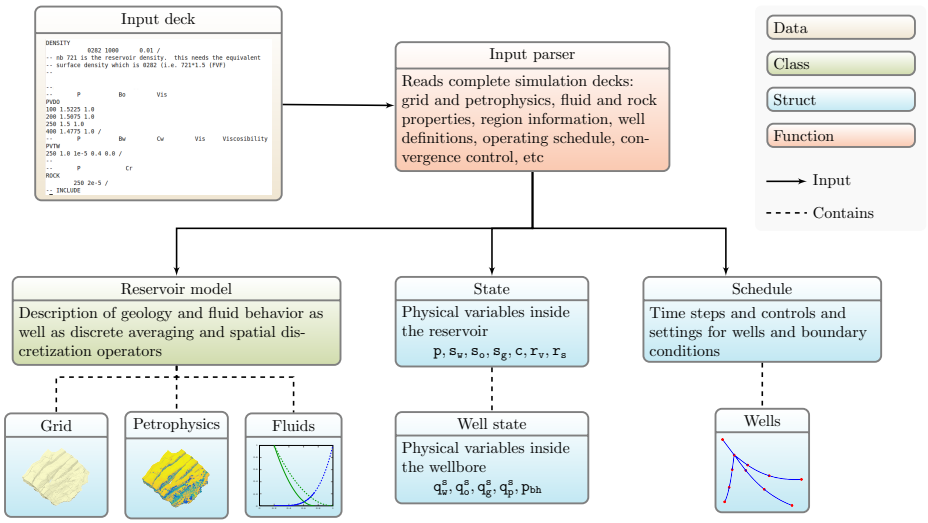


Fig. 8: By using the input parser that comes with the AD-OO (object oriented) framework, one can instantiate the data structures necessary to run a simulation from an industry-standard input deck.

used as a starting point to write simulators that are fit for purpose. For completeness, we will here go through one example of such a script. To set up a simulation, we need to construct three different data objects as illustrated in Figure 8: a class object describing the physical model, a structure containing variables describing the reservoir state, and a structure containing the schedule that specifies controls and settings on wells and boundary conditions and how these vary with time. Here, we assume that we have a reservoir model described in terms of an industry-standard input file [22]. We can then use functionality from the `deckformat` module, to read, interpret, and construct the necessary data objects from this input data file. We start by reading all keywords and data from the file.

```
deck = readEclipseDeck(file);
deck = convertDeckUnits(deck);
```

The data can be given in various types of units, which need to be converted to the standard SI units used by MRST. We then construct the three data structures that make up the physical model:

```
G     = computeGeometry(initEclipseGrid(deck));
rock  = compressRock(initEclipseRock(deck), G.cells.indexMap);
fluid = initDeckADIFluid(deck);
```

By convention, all grid constructors in MRST only output the information necessary to represent an unstructured grid and do not process this information to compute geometric information such as cell volumes, face areas, face normals, etc. However, as we have seen above, we need this information to compute transmissibilities and pore volumes and hence we also call a routine that computes this information. Similarly, the input parser for petrophysical data outputs values for all cells in the model and hence needs to be passed on to a routine that eliminates

data in cells set to be inactive. Having obtained all the necessary data, we can then call the constructor of the appropriate model object:

```
model = ThreePhaseBlackOilPolymerModel(G, rock, fluid, 'inputdata', deck);
```

Next, we instantiate the state object and set the initial conditions inside the reservoir. Here, we assume that the reservoir is initially in hydrostatic equilibrium, so that saturations/masses are defined to balance capillary pressure forces. This is a standard black-oil routine that works with three or fewer phases and is not aware of extra fluid components. The initial polymer concentration therefore needs to be specified manually.

```
state = initEclipseState(G, deck, initEclipseFluid(deck));
state.c    = zeros(G.cells.num,1);
state.cmax = state.c;
```

To set up the schedule, we need to know information about reservoir model, and hence this data object is constructed last,

```
schedule = convertDeckScheduleToMRST(model, deck);
```

Having established the necessary input data, we select the linear and nonlinear solvers.

```
nonlinearsolver = getNonLinearSolver(model, 'DynamicTimesteps', false, ...
                                             'useCPR', false);
nonlinearsolver.useRelaxation = true;
```

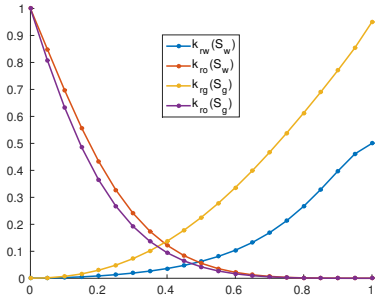
Here, we have set the simulator to use a simple relaxation procedure to stabilize the Newton iterations and MATLAB's standard `mldivide` as linear solver. By setting `useCPR` to be `true`, function `getNonLinearSolver` will set up a CPR preconditioner with either `BackslashSolverAD` based on `mldivide` or `AGMGSolverAD` with the AGMG multigrid solver [19]. (The main advantage of AGMG compared with other multigrid solvers is that it has a simple MATLAB interface and can be used directly as a drop-in replacement for `mldivide` without any data conversion or parameter tuning.) For small cases, `mldivide` can be efficient enough or even faster, while CPR preconditioned linear solver is typically more efficient or required for bigger cases. The option `DynamicTimesteps` being `false` says that we do not make any attempt at optimizing the time steps and only perform a standard Appleyard chop to cut time steps if the nonlinear solver fails to converge. If we turn on dynamic time stepping, the simulator will try to dynamically adjust the time steps to stay close to a targeted number of nonlinear iterations per time step.

We now have all that is necessary to run a simulation and can do this by calling the following script:

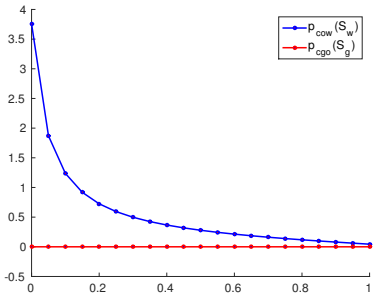
```
[wsols, states] = simulateScheduleAD(state, model, schedule, ...
                                     'NonLinearSolver', nonlinearsolver);
```

To visualize the output of the simulation, we invoke two graphical user interfaces from the `mrst-gui` module that let us view the reservoir variables and the well responses at all instances in time specified in the schedule.

```
plotToolbar(G, states); plotWell(G,schedule.control(1).W); view(3); axis tight
plotWellSols(wsols)
```



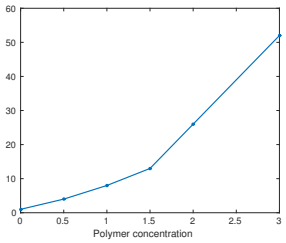
(a) Relative permeabilities



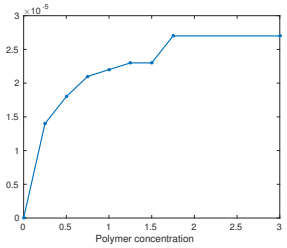
(b) Capillary pressure

	Water	Oil	Gas
Surface Density	1033 kg/m^3	860 kg/m^3	0.853 kg/m^3

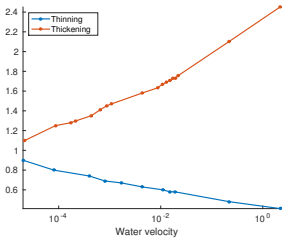
(c) Fluid properties



(d) Polymer viscosity multiplier



(e) Polymer adsorption



(f) Shear effects

Fig. 9: Properties and functions entering the fluid models.

As an alternative to using these two GUIs, well curves can be extracted to standard MATLAB arrays using either `getWellOutput` or `wellSolToVector`.

5 Numerical Examples

In this section, several examples are presented to demonstrate the validity and performance of the developed model. The first two examples verify the model and implementation against a leading commercial simulator [22]. Also, the effects of polymer injection and the impact of non-Newtonian fluid rheology on the water-flooding process are investigated. In the third example, we illustrate how the simulator is readily applicable to fully unstructured grids, whereas the last example uses the geological model of a real field to set up a test case with a high degree of realism. For simplicity, we use the same basic fluid model for the Example 1, 2 and 4, as summarized in Figure 9. For polymer, the dead pore space is set to 0.05, the residual reduction factor is 1.3, and the polymer is fully mixed into the aqueous phase (i.e., $\omega = 1$). For Example 3, we use a slightly different fluid model but the same polymer parameters. Link to complete codes for these examples can be found in `http://www.sintef.no/mrst/ad-eor/`.

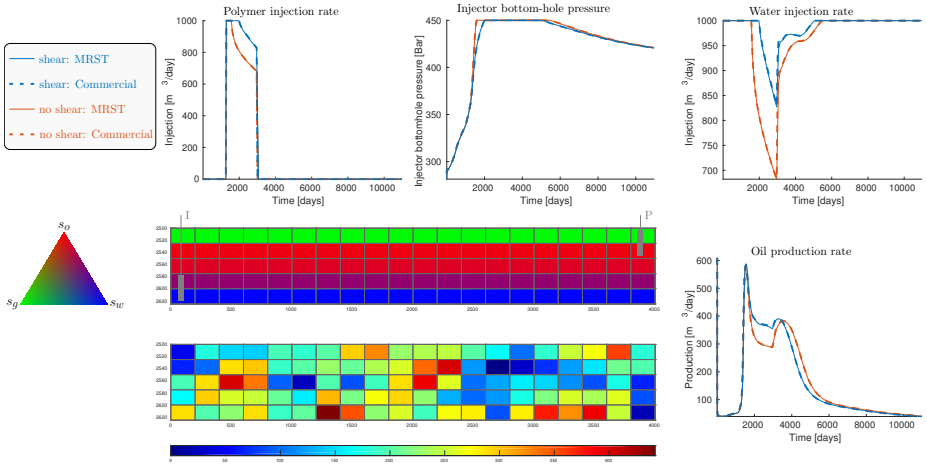


Fig. 10: Verification of the MRST solver against a commercial simulator for a 2D Cartesian example. Here, water is colored blue, oil is red, and gas is green.

5.1 Example 1: Verification against commercial simulator

A 2D example with heterogeneous porosity and permeability (Figure 10) is designed for the development and verification of the present model. In this example, the dimensions of the grid is $20 \times 1 \times 5$. The size of the domain is 4000 m $\times$ 200 m $\times$ 125 m. One injection well is located in the bottom two layers and one production well is located in the top two layers. Hydrostatic equilibration is used for initialization.

The polymer injection schedule follows a typical polymer waterflooding strategy. As shown in Figure 10, the flooding process begins with the primary water flooding (1260 days). Then a 1700-day polymer injection process with concentration 1 kg/m^3 is performed. Water injection is continued after the polymer injection. The injection well is under rate control with target rate 1000 m^3/day and upper limit of 450 bar on the bottom-hole pressure (bhp), whereas the production well is under pressure control with target bottom-home pressure 260 bar.

For comparison, two groups of simulations are performed with MRST and the commercial simulator. The first does not include shear effects (brown lines in Figure 10), and in the other one, shear effect is taken into consideration (blue lines in Figure 10). The results from MRST are indicated with solid lines and the results from the commercial simulator with dashed lines. From the results, it can be observed that the bottom-hole pressure for the injection well increases drastically when the polymer injection starts. When the bottom-hole pressure reaches the upper limit, the injection well switches to bottom-hole pressure control and the water injection rate drops rapidly. This can be explained in a natural way with the employed well model (3). The dissolution of the polymer increases the viscosity of the injecting water and as a result, the mobility of the water phase is decreased. According to (3), higher bottom-hole pressure is required to maintain the target injection rate. When the bottom-hole pressure reaches the limit, the water injection rate will drop rapidly as a result of the decreased mobility.

As shown in Figure 10, the results from MRST and the commercial simulator agree well. Abrupt changes when the polymer injection begins and ends are captured accurately. Both simulators predict the same shear-thinning behavior, which significantly improves injectivity and results in a much higher water rate during polymer injection.

5.2 Example 2: Sector model

In this example, we consider 3D synthetic sector model generated with MRST. The model has a physical extent of 1000 m × 675 m × 212 m, contains four vertical faults that intersect in the middle of the domain, and is represented on a $30 \times 20 \times 6$ corner-point grid, in which 2778 cells are active. There is one injection well located in the center of the sector and is perforated in the bottom three layers, and two production wells located to the east and west and perforated in the top layers; see Figure 11. The injector is under rate control with target rate 2500 m^3/day and bottom-hole pressure limit 290 bar, whereas the producers are under bottom-hole pressure control with target pressure 230 bar.

To investigate the effects of the polymer injection and different types of fluid rheology on the waterflooding process, four different simulations are performed with both MRST and the commercial simulator. The first simulations describe pure waterflooding. The second simulations describe polymer injection, but do not account for non-Newtonian fluid rheology during the injection process. The third and the fourth simulation setups assume that the polymer exhibits shear-thinning and shear-thickening behavior, respectively. Figure 11 reports water rate and bottom-hole pressure in the injector and water cut in the two producers.

For pure waterflooding, the bottom-hole pressure decays fast to a level below 250 bar during the first 150 days and then starts to gradually increase after approximately 400 days to maintain the specified injection rate until the end of simulation. In the polymer-flooding scenarios, the injectivity decreases dramatically once the polymer injection begins because of the increased viscosity of the polymer–water mixture. If the diluted polymer behaves like a Newtonian fluid, the bottom-hole pressure will quickly reach the upper limit and force the injector to switch from rate to pressure control, which in turn causes an immediate drop in the injection rate. As a result, the oil production declines during the injection of the polymer slug, but increases significantly during the tail production in both producers. Likewise, we see delayed water production in both producers. In the case of shear-thinning fluid rheology, the bottom-hole pressure also increases rapidly, but manages to stay below the bhp limit, which means that the targeted injection rate can be maintained. As a result, we maintain the initial oil production and achieve a better tail production as a result of improved displacement efficiency and volumetric sweep. When polymer with shear-thickening rheology is injected, the injectivity is drastically reduced, and in this case, the commercial simulator was not able to finish the simulation. A few report steps after the polymer injection starts, the simulator computes bottom-hole pressure values that are not well-behaved, which causes it to stop. MRST, on the other hand, manages to finish the simulation despite the somewhat unphysical setup.

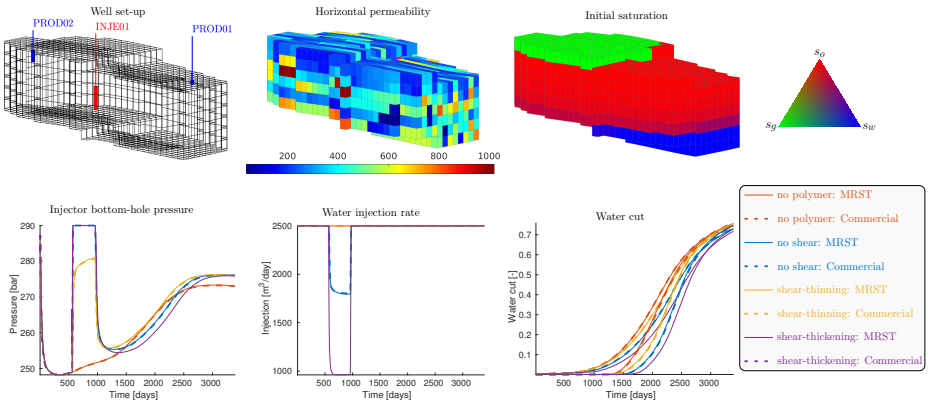


Fig. 11: Verification of the MRST polymer simulator against a commercial simulator with a 3D synthetic example with different types of polymer injected.

Table 1: Fluid densities and viscosities used in Example 3.

	Water	Oil	Gas
Compressibility	—	10^{-4} /bar	10^{-3}/bar
Viscosity	1 cP	5 cP	0.2 cP
Density (at 250 bar)	1033 kg/m^3	860 kg/m^3	400 kg/m^3

5.3 Example 3: Unstructured grids

The main purpose of the third example is to demonstrate that the polymer simulator is capable of simulating grids with general polyhedral geometries. To this end, we consider a vertical cross-section of a reservoir with dimensions of 1000×100 meters. There is an injector-producer pair included, where the producer has a curved well trajectory spanning a relatively large region of the reservoir. We consider a scenario in which one pore volume of water is injected over five years, a polymer slug added to the initial 7.5 months, and fluids are produced at a fixed bottom hole pressure of 250 bar. Table 1 lists fluid densities and viscosities.

To represent the reservoir, we consider four different grids: a fine Cartesian grid with 20 000 cells, a coarse Cartesian grid with 231 cells, an unstructured perpendicular bisector (PEBI) grid with 1966 cells refined around the wells, and a composite grid in which the coarse Cartesian grid is refined locally around the wells by adding Voronoi cells, giving in total 921 cells. The two latter grids were constructed using a new module in MRST for generating 2D and 3D Voronoi grids with cell centers and/or cell faces conforming to geological structures like well paths, faults, fractures, etc. The grid factory routines handle intersection of multiple faults, intersections of wells and faults, and faults intersecting at sharp angles; see [1, 5]. Figure 12 shows the four grids, along with snapshots of the phase saturations partway into the simulation. (In our color convention, water is blue, oil is red, and gas is green. The convention also applies to other examples in the paper.) Well responses are plotted in Figure 13.

Comparing the results from all four grids, we see that there are significant differences in the predicted injector bottom-hole pressures. The injector is best

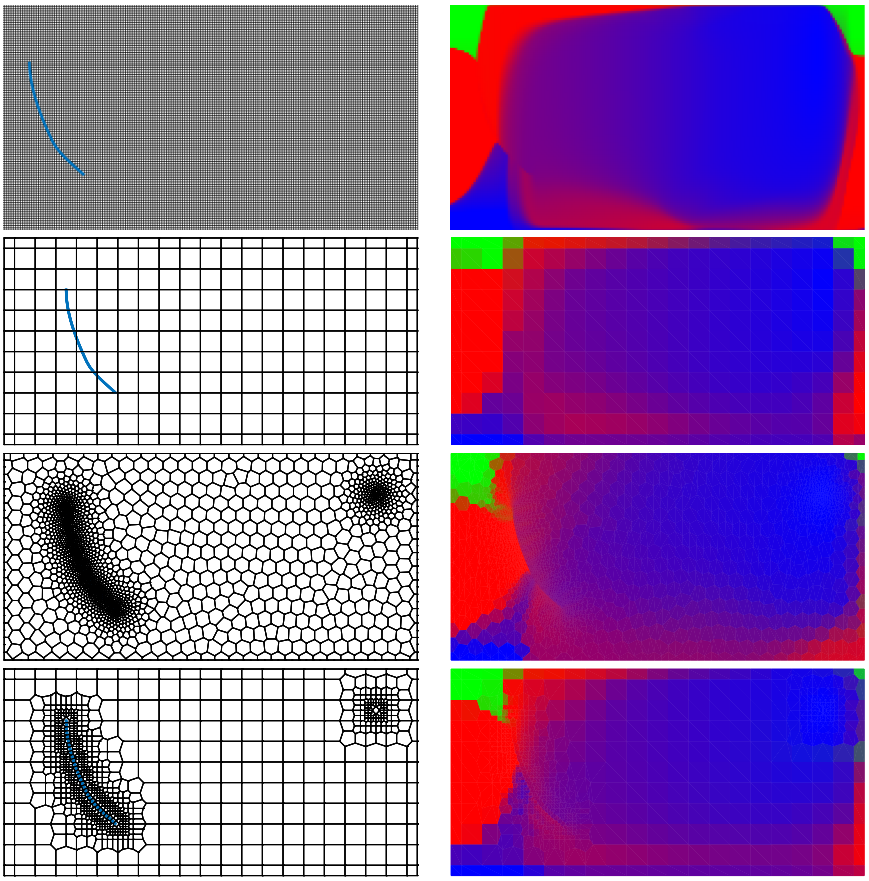
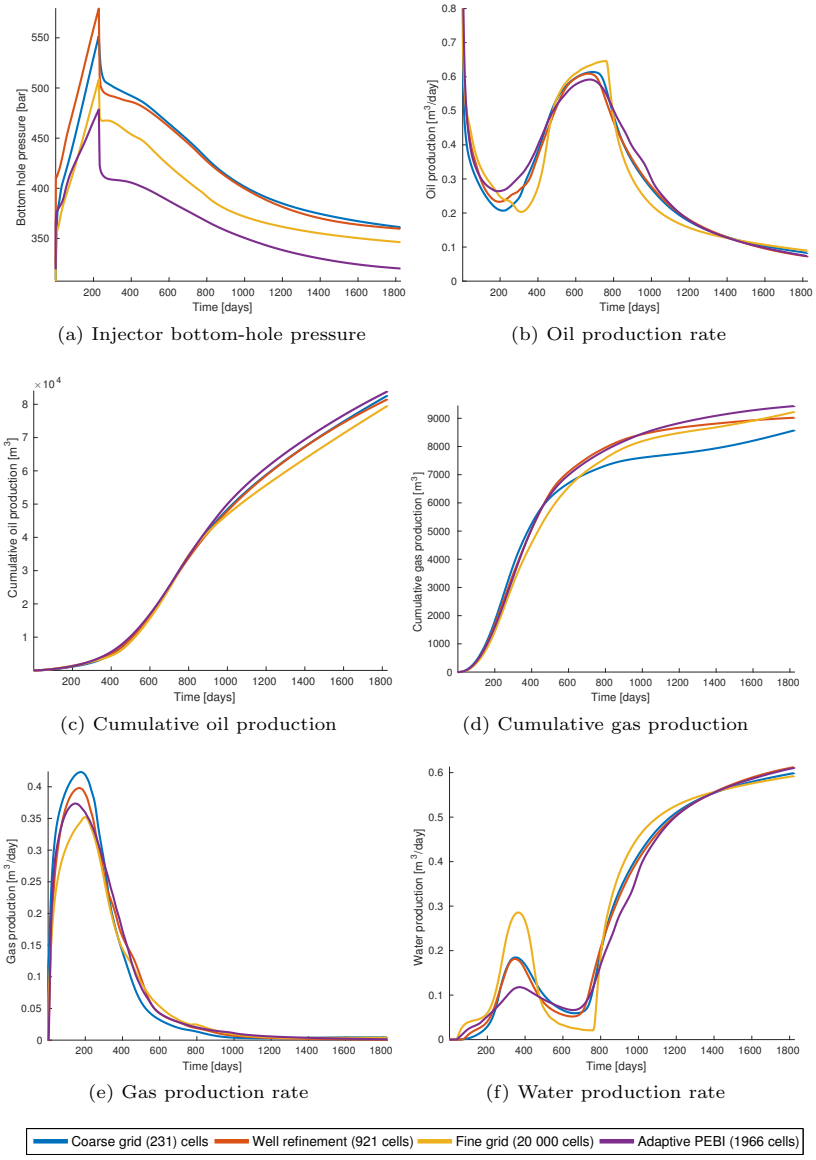


Fig. 12: The four grids considered in Example 3, along with the phase saturations computed after the injection of 0.60 PVI.

approximated by the PEBI grid, which uses 88 cells that align locally with the curved well path. The fine Cartesian grid approximates the well path in a stair-stepped manner using 74 cells, which on average are twice as large as the perforated PEBI cells. The composite grid also adapts to the well path, but here the 30 well cells are less regular and on average nine times larger than in the PEBI grid. Interestingly, the resulting bottom-hole curve is not significantly different from the coarse Cartesian grid, which only has 8 perforated cells. One possible explanation is that the default well indices computed by Peacemann's formula are only strictly correct for hexahedral cells. Likewise, since the initial fluid distribution is computed by sampling at the cell centroids, the oil–water contact is sharply resolved only on the fine Cartesian grid and is non-flat on the two unstructured grids.

Looking at the oil production, we see that all three coarse grids predict a too rapid initial decay and smooth the subsequent buildup and decay compared with the fine Cartesian grid. The result is that all three coarse grids slightly overpredict the cumulative oil production. The coarse Cartesian grid gives the largest



(a) Injector bottom-hole pressure

(b) Oil production rate

(c) Cumulative oil production

(d) Cumulative gas production

(e) Gas production rate

(f) Water production rate

Fig. 13: Well curves for the structured and unstructured grids of Example 3.

deviations in gas production, whereas the unstructured PEBI grid has the largest deviation in water rate. Altogether, it seems like the composite grid gives the closest match with the fine Cartesian grid.

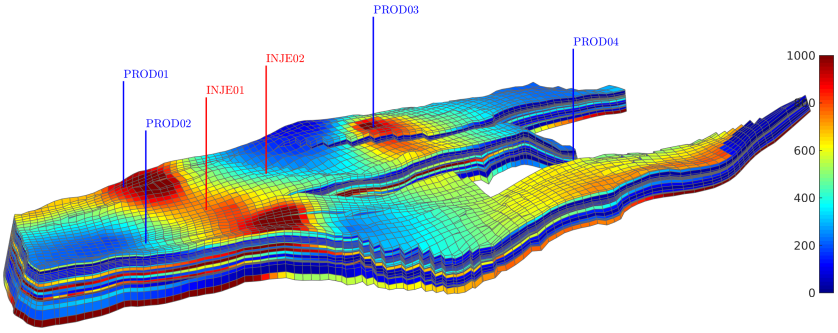
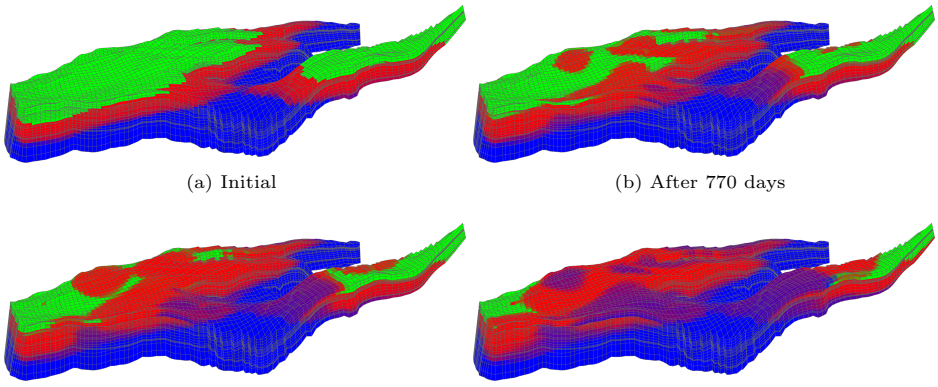


Fig. 14: Geological model and well setup for the Norne example. Colors show the horizontal permeability. To better distinguish zones of high and low permeability, the color axis is set to be between 0 and 1000 md; the actual permeability values extend to 3500 md.



(a) Initial

(b) After 770 days

(c) After 1170 days

(d) Final, after 2260 days

Fig. 15: Saturation distribution at different times for the Norne simulation.

5.4 Example 4: Norne

Norne is an oil field located in the Norwegian Sea. The simulation model of this field has recently been made publicly available as an open data set [20]. Here, we will use a slightly simplified version of this simulation model, in which we have removed one tiny disconnected region and disabled some features related to flux regions, fault multipliers, equilibration regions, and so on. Furthermore, we replace the fairly complicated well-control schedule representing the real field history with six wells operating under simpler control schedules. To run this example, the AGMG multigrid solver [19] is required.

The simulation model consists of 44915 active cells and has a total pore volume of $8.16 \cdot 10^8$ m^3. The two injection wells (shown with red color in Figure 14) are under rate control with target rate 30000 m^3/day and a bottom-hole pressure limit of 600 bar. Four production wells (shown with blue color in Figure 14) are

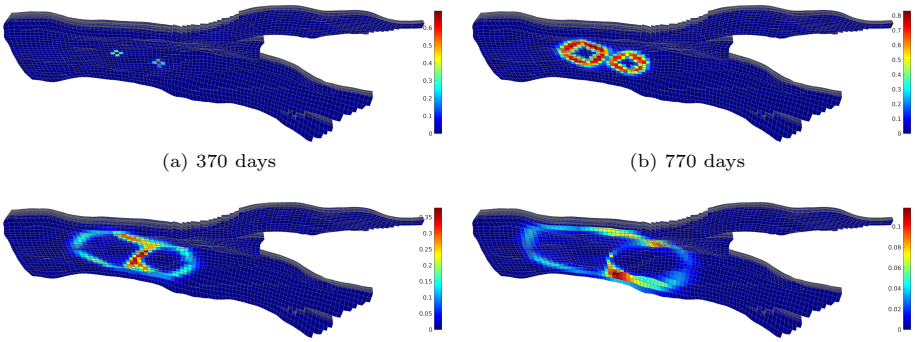
under bottom-hole pressure control with target pressure 200 bar, 200 bar, 190 bar, and 180 bar, respectively. The injection begins with primary waterflooding for 360 days. Then, polymer with concentration 1 kg/m^3 is injected for 340 days. Waterflooding continues for another 1460 days after the polymer injection stops. The total simulation time covers a period of 2260 days. Non-Newtonian fluid rheology is not considered in this example.

The initial saturation is initialized with hydrostatic equilibration (Figure 15a). The saturation distribution and polymer concentration at different times are shown in Figure 15 and Figure 16, respectively. The evolution of water injection rate, bottom-hole pressure in injection wells, oil production rate, and water cut are reported in Figure 17. For comparison, the resulting well curves for a pure waterflooding scenario are plotted as dashed lines. The impact of polymer injection on the injection process, like injectivity, injection rate, and water cut is clearly shown through the resulting well curves. The main effect of polymer is that the reduced injectivity leads to a shift in the oil rate, which diminishes the overall oil production. With a short time horizon of 2260 days, the suggested polymer injection is not a good engineering solution. We emphasize that polymer injection is not performed in reality on Norne, and the polymer scenario studied herein is invented by the authors for illustration purposes.

Overall, our artificial polymer flooding scenario represents a computationally challenging problem, and not surprisingly, the implicit solver struggles to converge for some of the time steps. However, use of adaptive chopping of time steps makes the simulator more robust and enables it to run through the specified simulation schedule. MRST offers both reactive and predictive time-step control, similar to those seen in many commercial simulators. The reactive part uses upper bounds on the number of iterations allowed. If any of these bounds are exceeded, the simulator will halve the time step, and continue to do so until the iteration bounds are not exceeded. If the time step is reduced below a given minimum, the simulator will stop and report convergence failure. Likewise, if the current step size has been successfully used a given number of times, the simulator will try to increase it by a given factor, and this is repeated until one reaches a given maximum time step. MRST can also set upper bounds on the absolute or relative changes one or more of the physical variables (typically saturation) are allowed to change during one iteration and use this to *predict* the time-step size. Another alternative is to set a target for the number of nonlinear iterations and let the simulator use the previous convergence history to guess the size of the time step that will ensure that the iteration target is met. If desired, all parameters controlling these strategies can be prescribed by the user. In addition, one can gradually ramp up the initial time step so that it increases geometrically towards a given target. Without these capabilities, manual modifications of the time steps and multiple reruns would likely have been necessary to get a simulation through. In this particular simulation, we used an initial ramp up specified in the schedule combined with the reactive strategy with an upper iteration bound of 15.

5.5 Example 5: Polymer flooding optimization with adjoint method

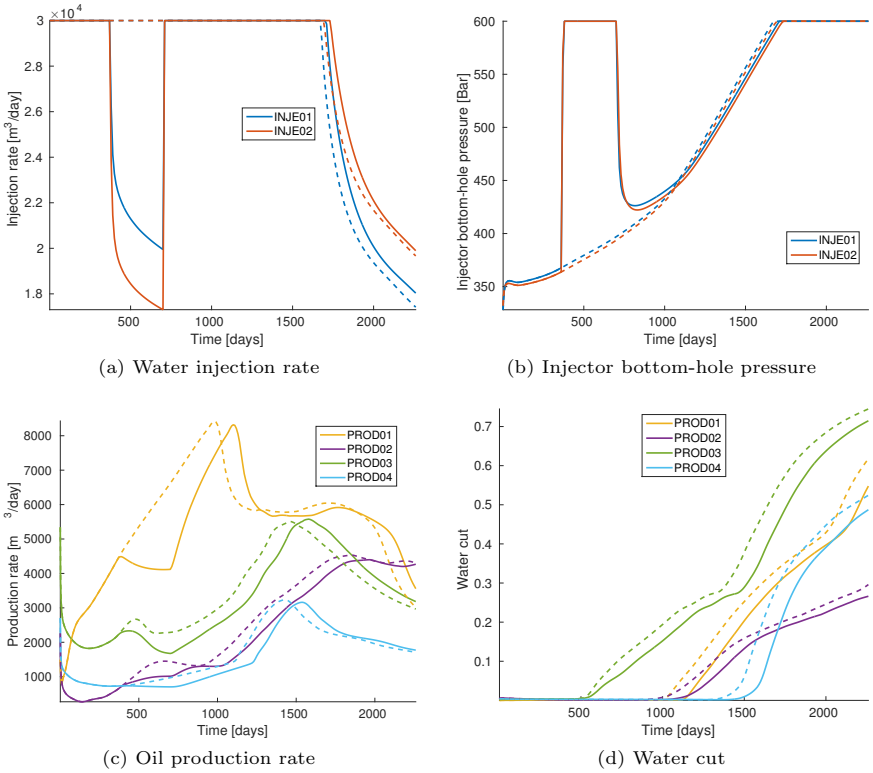
The main purpose of polymer injection is to increase the economics of the recovery process. To measure this, we consider the net present value, which accounts for the



(a) 370 days (b) 770 days

(c) 1170 days (d) Final time, 2260 days

Fig. 16: Polymer concentration at different times for the Norne simulation.



(a) Water injection rate (b) Injector bottom-hole pressure

(c) Oil production rate (d) Water cut

Fig. 17: Well curves for the Norne case. Dashed lines represent a pure waterflooding scenario and solid lines a polymer injection scenario.

Table 2: Prices used in the calculation of NPV (21).

	Prices (in US dollars)
Oil revenue	60 USD/stb
Gas revenue	2.8 USD/mmbtu
Polymer cost	5 USD/kg
Water injection cost	5 USD/stb
Water production processing cost	5 USD/stb
Yearly discount factor	0.05

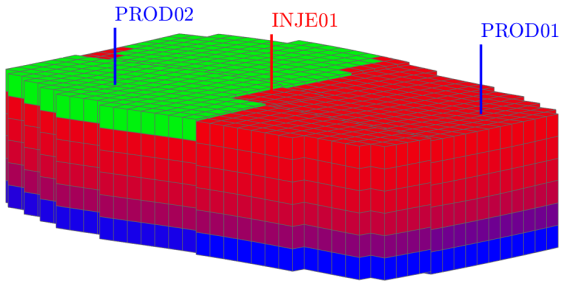


Fig. 18: Initial saturations of for the polymer optimization example Case 1.

production revenue of oil and gas, the cost related to the injection and production process, and the discount of value with time

$$NPV(T) = \int_{t=0}^{T} (r_o q_o + r_g q_g - (r_{iw} q_{iw} + r_w q_w + r_{ip} q_{ip}))(1+d)^{-t} dt. \qquad (21)$$

Here, r_o and r_g are the oil and gas revenue prices and q_o and q_g are the oil and gas production rates, respectively. As a result, $r_o q_o + r_g q_g$ represents the revenue due to production of oil and gas. Moreover, r_{iw}, r_w, and r_{ip} represent water injection cost, water production processing cost, and polymer cost, respectively, whereas d is the discount rate and q_{iw}, q_w and q_{ip} are water injection rate, water production rate, and polymer injection rate, respectively. Hence, $r_{iw} q_{iw} + r_w q_w + r_{ip} q_{ip}$ represents related costs during the polymer water-flooding and production process. The values employed in this section are listed in Table 2, and are invented by the authors for illustration purpose.

To maximize NPV, we will optimize polymer injection concentration. To this end, we will use a rigorous gradient-based mathematical optimization method, in which gradients of the NPV with respect to the current controls are computed using an adjoint formulation. Adjoint formulations is part of the AD-OO framework, and has previously been discussed e.g., in [15,12]. Specifically, we will use the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm [18], which is a quasi-Newton method, in combination with a line-search algorithm with termination criteria based on the Wolfe conditions [26].

We investigate the optimization of two water-flooding cases. For both cases, we increase the oil viscosity to be between 10 and 20 cp, while water viscosity remains 0.318 cp. No shear effect is considered during the simulation.

Case 1: We consider a synthetic 1200 m × 1000 m × 150 m sector model with four vertical faults intersecting in the middle of the domain similar as in

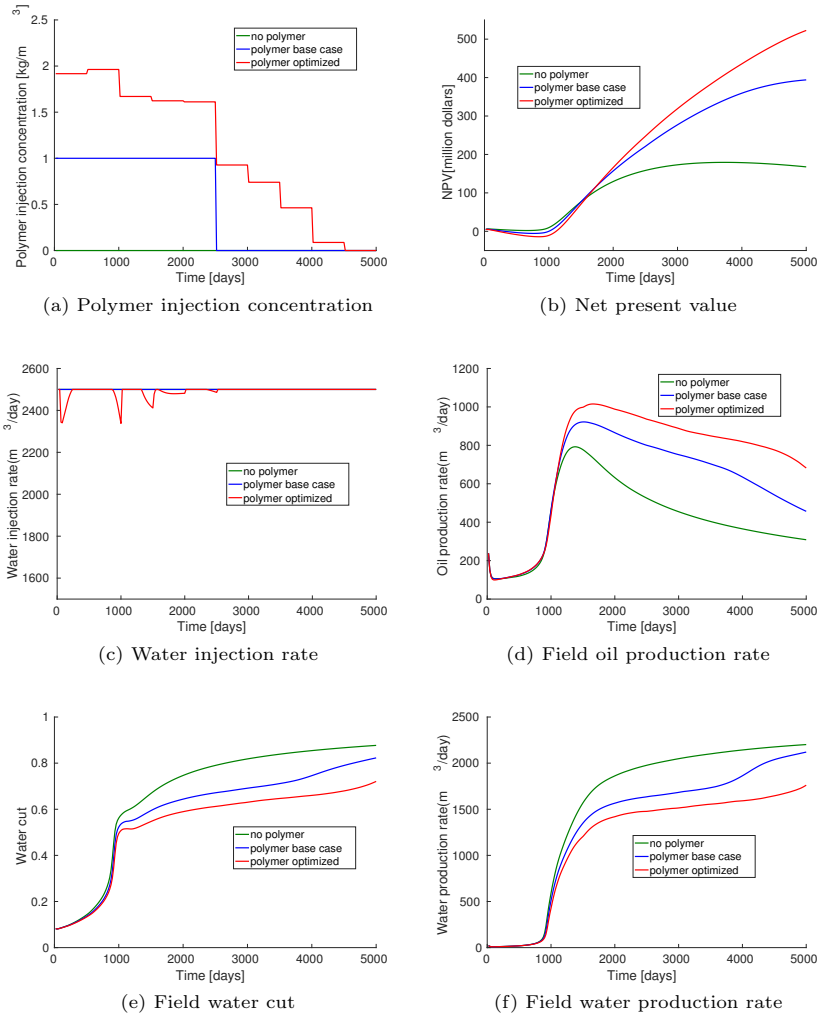
Section 5.2 (see Figure 18). The formation is represented on a $30 \times 25 \times 6$ corner-point grid with 3528 active cells. The injector is under rate control with target rate 2500 m^3/day and bottom-hole pressure limit 600 bar, whereas the producers are under bottom-hole pressure control with target pressure 230 bar. The total flooding process is 5000 days. To optimize, we split the schedule to ten periods of 500 days and try to find the polymer injection concentration for each period that maximizes overall NPV (21). The maximum available polymer concentration is 2.5 kg/m^3.

Results from three different flooding processes are shown in Figure 19. Green lines represent pure water flooding, whose NPV curve starts to flatten around 3000 days and reaches its peak net-present value after approximately 3600 days (see Figure 19b). After this time, the economic value decays, mostly due to the high water cut (Figure 19e) and low oil production rate (Figure 19d). The blue lines represent a straightforward polymer injection strategy, in which 1 kg/m^3 polymer is injected for the first half of the total flooding procedure and pure water flooding for the second half flooding process. This improves the NPV of the whole flooding operation (Figure 19b). From Figures 19d and 19f we see that polymer not only improves the oil production rate, but also reduces water production. The flooding procedure will be the starting point of the optimization process, and is referred to as the base case.

Red lines in Figure 19 represent the optimized flooding process. Compared with the base case, more oil is produced and the water production rate is decreased further, which means less cost related to water production. From Figure 19a, we see that the optimization program suggests a relatively high polymer concentration at the beginning, lower polymer injection concentration later, and no polymer injection for the last period. When flooding with higher polymer concentration, it is not suggested to use the highest possible polymer concentration (2.5 kg/m^3). Instead, it is suggested to use the highest possible concentration that maintains the water injection rate around the target rate (2500 m^3/day), which implies that maintaining the water injection rate in this scenario is important for achieving optimal NPV from polymer flooding operation. Notice also that the bottom-hole pressure is kept around its upper limit when injecting with higher polymer concentration.

Figure 20a shows a breakdown of NPV into the five terms from (21), i.e., revenue from oil and gas production and cost from water injection, water production, and use of polymer. Likewise, Figure 20b shows the breakdown of the relative increase in NPV and polymer cost from the base case to the optimized case, and how this is balanced by increased revenue from oil production and decreased costs for water injection and production (difference in revenue from gas production was negligible). From the breakdown, we can see, the increase in oil production and reduction in water production play the major role in achieving higher NPV.

Case 2: We use the same grid as in Section 5.2 (Figure 11) with the same well controls as in Case 1. Due to smaller size of the formation and less oil in place, we change the flooding period to be 3000 days, which is split into ten even periods. The results without polymer injection, the base polymer injection procedure, and the optimized one are shown in Figure 21. The optimized polymer injection gives higher NPV than the base case, which in turn is better than pure water flooding. Optimization also suggests higher polymer injection concentration in the beginning and lower polymer injection concentration for later. However, different from Case 1, maximum polymer concentration is used initially (Figure 21a), which

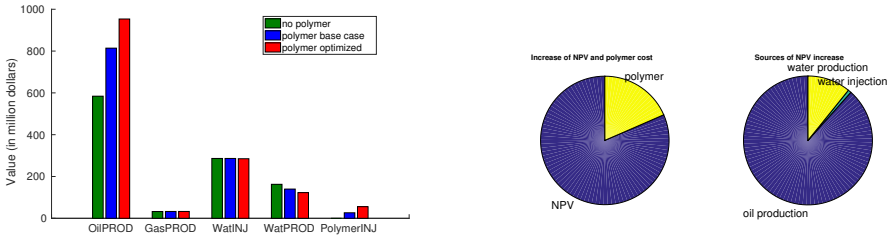


(a) Polymer injection concentration (b) Net present value

(c) Water injection rate (d) Field oil production rate

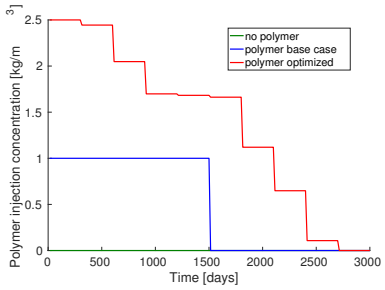
(e) Field water cut (f) Field water production rate

Fig. 19: The polymer injection schedule, NPV curves, and well curves for Case 1.

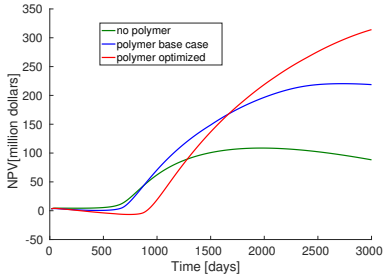


(a) Breakdown of NPV (21) (b) Difference from base to optimzed case

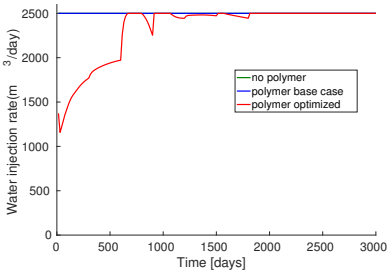
Fig. 20: Breakdown of NPV for the three different flooding processes, and changes in revenues and costs from base case to optimized flooding for Case 1.



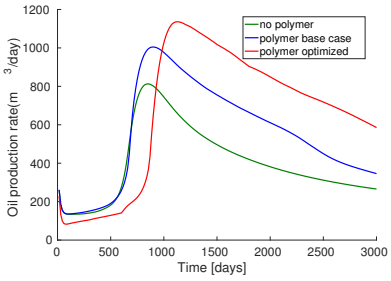
(a) Polymer injection concentration



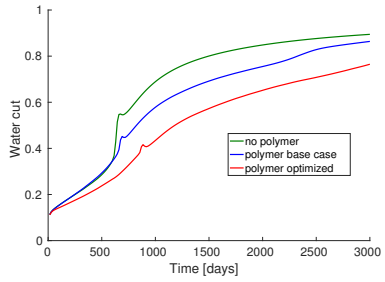
(b) Net present value (NPV)



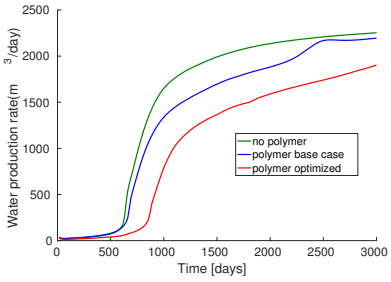
(c) Water injection rate



(d) Field oil production rate

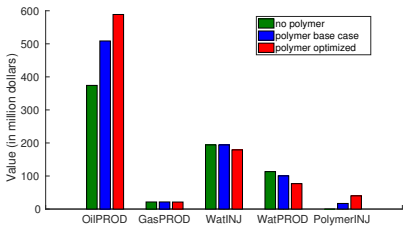


(e) Field water cut

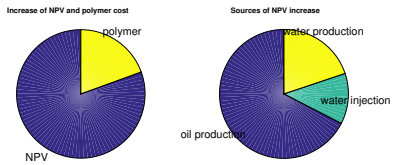


(f) Field water production rate

Fig. 21: The polymer injection schedule, NPV curves, and well curves for for Case 2.



(a) Breakdown of NPV (21)



(b) Difference from base to optimzed case

Fig. 22: Breakdown of NPV for the three different flooding processes, and changes in revenues and costs from base case to optimized flooding for Case 2.

reduces both the water injection rate (Figure 21c) and NPV (Figure 21b) during the first three periods (900 days). Breakdown in the increase of NPV and polymer costs from the base case to the optimized one is shown in Figure 22b. Compared with Case 1 (Figure 20b), the relative increase in oil production is smaller, while a larger fraction of the NPV increase can be attributed to a decrease in water injection.

6 Concluding Remarks

In an earlier paper [11], we presented the free, open-source MRST software, which has later become a community code and is used by many researchers within the computational geosciences.

Recently, the software has been modernized with several new features such as discrete differential operators, automatic differentiation, and an object-oriented programming framework, which contribute to make MRST a perfect platform for fast development and prototyping simulators capable of running industry-grade simulations. Herein, we have discussed in detail how this framework can be utilized to develop a flexible simulator for polymer flooding, whose main intent is to serve as a research tool for developing new models and computational methods for simulating water-based EOR processes. To enable other researchers to benefit from our work, we have described key components of MRST in some detail and discussed the key steps necessary to extend an existing black-oil simulator to polymer flooding, including effects such as viscosity enhancement, adsorption, inaccessible pore space, permeability reduction, and non-Newtonian fluid rheology. The resulting simulator is released as part of a new EOR module in MRST (`ad-eor`), which also includes a few surfactant models. Using the flexible platform design of MRST, we believe that it is not very difficult to extend the capabilities of the `ad-eor` module to models with similar flow physics, including surfactant-polymer, alkali-surfactant-polymer, etc.

To prove the validity of the polymer simulator, we have benchmarked it against a leading commercial simulator and shown that it produces virtually identical results for two test cases in 2D and 3D, including three fluid phases, water flooding or polymer flooding, with and without shear effects. Flexibility with respect to different grids was demonstrated in a test case involving unstructured grids with polyhedral cell geometries. We also showed that the simulator is capable of handling industry-relevant simulations by posing a polymer flooding scenario on a model with reservoir geometry and petrophysics of a real oil and gas field. Finally, we utilized the optimization module from MRST to optimize the polymer flooding process for two synthetic sector models, and discussed and analyzed differences in the resulting injection strategies. Evidence that the simulator framework is a good platform for testing new computational methods can also be found in [3]. Here, the framework is used to develop a new and efficient multiscale method for polymer flooding relying on a sequentially implicit formulation instead of the fully implicit formulation described herein.

7 Acknowledgments

The work has been funded in part by the Research Council of Norway under grant no. 244361. The authors want to thank Statoil (operator of the Norne field) and its license partners ENI and Petoro for the release of the Norne data. Further, the authors acknowledge the IO Center at NTNU for coordination of the Norne cases and Statoil for releasing the simulation model under an open data licence as part of the Open Porous Media (OPM) initiative. We also appreciate helpful discussions and suggestions from Stein Krogstad (SINTEF) regarding the polymer optimization examples.

References

1. Berge, R.L.: Unstructured PEBI grids adapting to geological features in subsurface reservoirs. Master's thesis, Norwegian University of Science and Technology (2016)
2. Gries, S., Stüben, K., Brown, G.L., Chen, D., Collins, D.A.: Preconditioning for efficiently applying algebraic multigrid in fully implicit reservoir simulations. SPE J. **19**(04), 726–736 (2014). DOI 10.2118/163608-PA
3. Hilden, S.T., Møyner, O., Lie, K.A., Bao, K.: Multiscale simulation of polymer flooding with shear effects. Transp. Porous Media **113**(1), 111–135 (2016). DOI 10.1007/s11242-016-0682-2
4. Jansen, J.D.: Adjoint-based optimization of multi-phase flow through porous media – a review. Computers & Fluids **46**(1, SI), 40–51 (2011). DOI 10.1016/j.compfluid.2010.09.039
5. Klemetsdal, Ø.S., Berge, R.L., Lie, K.A., Nilsen, H.M., Møyner, O.: Unstructured gridding and consistent discretizations for reservoirs with faults and complex wells. In: SPE Reservoir Simulation Conference, Montgomery, Texas, USA, 20-22 February 2017 (2017). DOI 10.2118/182679-MS
6. Krogstad, S., Lie, K.A., Møyner, O., Nilsen, H.M., Raynaud, X., Skaflestad, B.: MRST-AD – an open-source framework for rapid prototyping and evaluation of reservoir simulation problems. In: SPE reservoir simulation symposium. Society of Petroleum Engineers (2015). DOI 10.2118/173317-MS
7. Lake, L.W.: Enhanced Oil Recovery. Prentice-Hall (1989)
8. Li, W., Dong, Z., Sun, J., Schechter, D.S.: Polymer-alternating-gas simulation: A case study. In: SPE EOR Conference at Oil and Gas West Asia. Society of Petroleum Engineers (2014). DOI 10.2118/169734-MS
9. Li, Z., Delshad, M.: Development of an analytical injectivity model for non-Newtonian polymer solutions. In: SPE Reservoir Simulation Symposium, 18-20 February, The Woodlands, Texas, USA (2013). DOI 10.2118/163672-MS. SPE-163672-MS
10. Lie, K.A.: An Introduction to Reservoir Simulation Using MATLAB: User guide for the Matlab Reservoir Simulation Toolbox (MRST). SINTEF ICT, http://www.sintef.no/Projectweb/MRST/publications (2016)
11. Lie, K.A., Krogstad, S., Ligaarden, I.S., Natvig, J.R., Nilsen, H.M., Skaflestad, B.: Open-source MATLAB implementation of consistent discretisations on complex grids. Comput. Geosci. **16**(2), 297–322 (2012). DOI 10.1007/s10596-011-9244-4
12. Lie, K.A., Møyner, O., Krogstad, S.: Application of flow diagnostics and multiscale methods for reservoir management. In: SPE Reservoir Simulation Symposium. Society of Petroleum Engineers (2015). DOI 10.2118/173306-MS
13. Littmann, W.: Polymer flooding. Elsevier (1988)
14. Luo, H.S., Delshad, M., Li, Z.T., Shahmoradi, A.: Numerical simulation of the impact of polymer rheology on polymer injectivity using a multilevel local grid refinement method. Petrol. Sci. pp. 1–16 (2015). DOI 10.1007/s12182-015-0066-1
15. Møyner, O., Krogstad, S., Lie, K.A.: The application of flow diagnostics for reservoir management. SPE Journal **20**(02), 306–323 (2015). DOI 10.2118/171557-PA
16. MRST: The MATLAB Reservoir Simulation Toolbox. www.sintef.no/MRST (2016b)
17. Neidinger, R.D.: Introduction to automatic differentiation and MATLAB object-oriented programming. SIAM Review **52**(3), 545–563 (2010). DOI 10.1137/080743627
18. Nocedal, J., Wright, S.: Numerical optimization. Springer Science & Business Media (2006)

19. Notay, Y.: An aggregation-based algebraic multigrid method. Electron. Trans. Numer. Anal. **37**, 123–140 (2010)
20. Open Porous Media initiative: Open datasets (2015). URL http://www.opm-project.org
21. Peaceman, D.W.: Interpretation of well-block pressures in numerical reservoir simulation. Soc. Petrol. Eng. J. **18**(3), 183—194 (1978). DOI 10.2118/6893-PA
22. Schlumberger: Eclipse Technical Description, Version 2013.2 (2013)
23. Sheng, J.J., Leonhardt, B., Azri, N.: Status of polymer-flooding technology. J. Canadian Petrol. Tech. **54**(02), 116–126 (2015). DOI 10.2118/174541-PA
24. Sorbie, K.S.: Polymer-Improved Oil Recovery. Springer Science & Business Media (1991)
25. Todd, M.R., Longstaff, W.J.: The development, testing, and application of a numerical simulator for predicting miscible flood performance. J. Petrol. Tech. **24**(07), 874–882 (1972). DOI 10.2118/3484-PA
26. Wolfe, P.: Convergence conditions for ascent methods. SIAM review **11**(2), 226–235 (1969). DOI 10.1137/1011036