Archive of Numerical Software
vol. 5, no. 1, 2017, pages 129–149
DOI: 10.11588/ans.2017.1.28068

Using Dune-Fem for Adaptive Higher Order Discontinuous Galerkin Methods for Two-phase Flow in Porous Media

Birane Kane[1]

[1]Institute of Applied Analysis and Numerical Simulation, University of Stuttgart

Received: February 17th, 2016; **final revision:** November 10th, 2016; **published:** March 6th, 2017.

Abstract: In this document we present higher order Discontinuous Galerkin discretization of a two-phase flow model describing subsurface flow in strongly heterogeneous porous media. The flow in the domain is immiscible and incompressible with no mass transfer between phases. We consider a fully implicit, locally conservative, higher order discretization on adaptively generated meshes. The implementation is based on the open-source PDE software framework `Dune`.

1 Introduction

Simulation of multi-phase flows and transport processes in porous media requires careful numerical treatment due to the strong heterogeneity of the underlying porous medium. The spatial discretization requires locally conservative methods in order to be able to follow small concentrations [6]. Discontinuous Galerkin (DG), Finite Volume and Mixed Finite Element, are examples of discretization methods which achieve local conservation at the element level [13]. The first DG method was originally developed for solving the neutron transport problem [19]. Since then, numerous DG methods have been developed for hyperbolic problems, the Bassi and Rebay [5] method and the Local Discontinuous Galerkin (LDG) method introduced in [11] are some examples among others. Independently of the development of the DG methods for hyperbolic equations, Interior Penalty (IP) Discontinuous Galerkin methods for elliptic and parabolic equations were introduced in [2], [4], [14], [23]. DG methods present attractive features such as an inherent local and global conservation, a high-order accuracy, a high parallel efficiency and a geometric flexibility (unstructured meshes and non-conforming grids) allowing an easier local *hp*-adaptivity. Furthermore, the ability of DG methods to treat rough coefficient problems and capture discontinuities in solutions allows them to be suitable candidates for the discretization of PDE's arising in Environmental Engineering.

Application of DG methods to incompressible two-phase flow started with [7], [18], [21]. The initial approach consisted in a decoupled formulation where first a pressure equation is solved implicitly and then the saturation is advanced by an explicit time stepping scheme (Implicit

Pressure Explicit Saturation). Upwinding, slope limiting techniques, and sometimes H(div) projection were required in order to remove unphysical oscillations and to ensure convergence. More recently, Bastian [8] presented a fully coupled symmetric interior penalty DG formulation for incompressible two-phase flow based on a formulation using a wetting-phase potential/capillary potential formulation. Discontinuity in capillary pressure functions is taken into account by incorporating the interface conditions into the penalty terms for the capillary potential. Heterogeneity in absolute permeability is treated by weighted averages. A higher-order diagonally implicit Runge-Kutta method in time is used and there is no post processing of the velocity or slope limiting. The author did not use any kind of adaptivity and only piecewise linear and piecewise quadratic functions are employed.

In this work, we implement and evaluate numerically Interior Penalty DG methods for 2d and 3d incompressible, immiscible, two-phase flow. We consider a strongly heterogeneous porous medium and discontinuous capillary pressure functions. We write the system in terms of a phase-pressure/phase-saturation formulation. Adams-Moulton schemes of first and second order in time are combined with various Interior Penalty DG discretizations in space such as the Symmetric Interior Penalty Galerkin (SIPG), the Nonsymmetric Interior Penalty Galerkin (NIPG) and the Incomplete Interior Penalty Galerkin (IIPG) [3]. This implicit space time discretization leads to a fully coupled nonlinear system requiring to build a Jacobian matrix at each time step for the Newton-Raphson method. We include in our implementation local mesh adaptivity on non-conforming grids. To our knowledge, this is the first time the concept of local h-adaptivity is incorporated in the DG discretization of a 3d two-phase flow with strong heterogeneity, discontinuous capillary pressure functions and gravity effects. The milestone contribution of Klieber & Rivière [18] restrained itself to a decoupled formulation with continuous capillary pressure functions and only 2d flow on non-conforming simplicials grids were considered. We use higher order polynomial degree up to piecewise cubics.

The rest of this document is organised as follows. In the next section, we describe the two-phase flow model. The DG discretization is introduced in section 3. The adaptive strategy in space is outlined in section 4. The implementation with `Dune-Fem` is described in section 5. Numerical examples are provided in section 6. Finally concluding remarks are provided in the last section.

2 Problem Setting

This section introduces the mathematical formulation of a two-phase Darcy problem modeling porous-media flow. The flow is immiscible and incompressible with no mass transfer between phases.

2.1 Two-phase flow model

We consider an open and bounded domain $\Omega \in \mathbb{R}^d$, $d \in \{1, 2, 3\}$ and the time interval $\mathcal{J} = (0, T)$, $T > 0$. The flow of the wetting-phase and the nonwetting-phase is described by the Darcy's law and the continuity equation for each phase, namely,

$$v_\alpha = -\lambda_\alpha K(\nabla p_\alpha - \rho_\alpha g), \tag{2.1}$$

$$\phi \frac{\partial \rho_\alpha s_\alpha}{\partial t} + \nabla \cdot (\rho_\alpha v_\alpha) = \rho_\alpha q_\alpha, \tag{2.2}$$

$$\sum_\alpha s_\alpha = 1, \tag{2.3}$$

$$p_n - p_w = p_c(s_{w,e}). \tag{2.4}$$

Here, we search for the phase pressures p_α and the phase saturations s_α, $\alpha \in \{w, n\}$. We denote with subscript w the wetting-phase and with subscript n the nonwetting-phase. K is the permeability of the porous medium, ρ_α is the phase density, q_α is a source/sink term and g is the constant gravitational vector. We assume the porosity ϕ is time independent and uniformly bounded from above and below; that is there exist $\phi_1, \phi_2 > 0$ such that:

$$0 < \phi_1 \leq \phi \leq \phi_2.$$

Phase mobilities λ_α are defined by

$$\lambda_\alpha = \frac{k_{r\alpha}}{\mu_\alpha}, \alpha \in \{w, n\}, \tag{2.5}$$

where μ_α is the phase viscosity and $k_{r\alpha}$ is the relative permeability of phase α. The relative permeabilities are functions that depend nonlinearly on the phase saturation (i.e. $k_{r\alpha} = k_{r\alpha}(s_\alpha)$). Models for the relative permeability are the van-Genuchten model [22] and the Brooks-Corey model [10]. For example, in the Brooks-Corey model,

$$k_{rw}(s_{w,e}) = s_{w,e}^{\frac{2+3\theta}{\theta}}, \quad k_{rn}(s_{n,e}) = (s_{n,e})^2(1 - (1 - s_{n,e})^{\frac{2+\theta}{\theta}}), \tag{2.6}$$

where the effective saturation $s_{\alpha,e}$ is

$$s_{\alpha,e} = \frac{s_\alpha - s_{\alpha,r}}{1 - s_{w,r} - s_{n,r}}, \quad \forall \alpha \in \{w, n\}. \tag{2.7}$$

Here, $s_{\alpha,r}$, $\alpha \in \{w, n\}$ are the phase residual saturations. The parameter $\theta \in [0.2, 3.0]$ is a result of the inhomogeneity of the medium. A highly heterogeneous porous medium is characterized by a large θ.

The capillary pressure $p_c = p_c(s_{w,e})$ is a function of the phase saturation. For the Brooks-Corey formulation,

$$p_c(s) = p_d s_{w,e}^{-1/\theta}. \tag{2.8}$$

Here, $p_d \geq 0$ is the constant entry pressure, needed to displace the fluid from the largest pore.

2.1.1 Wetting-phase-pressure/nonwetting-phase-saturation formulation

From the constitutive relations (2.3) and (2.4), we can rewrite the two-phase flow problem as a system of two equations with two unknowns p_w and s_n,

$$\begin{aligned} -\nabla \cdot (\lambda_t K \nabla p_w + \lambda_n K \nabla p_c - (\rho_w \lambda_w + \rho_n \lambda_n) K g) &= q_w + q_n, \\ \phi \frac{\partial s_n}{\partial t} - \nabla \cdot (\lambda_n K (\nabla p_w - \rho_n g)) - \nabla \cdot (\lambda_n K \nabla p_c) &= q_n. \end{aligned} \tag{2.9}$$

Here, $\lambda_t = \lambda_w + \lambda_n$ denotes the total mobility. The first equation of (2.9) is of elliptic type with respect to the pressure p_w. The type of the second equation of (2.9) is either nonlinear hyperbolic if $\frac{\partial p_c(s_n)}{\partial s_n} \equiv 0$ or degenerate parabolic if the capillary pressure is not neglected. The diffusion term might degenerate if $\lambda_n(s_n = 0) = 0$.

2.1.2 Boundary properties

In order to have a complete system we add appropriate boundary and initial conditions. Thus, we assume that the boundary of the system is divided into disjoint open sets $\partial\Omega = \bar{\Gamma}_D \cup \bar{\Gamma}_N$. We denote by n the outward normal to $\partial\Omega$.

$$s_n(x, 0) = s_n^0(x), p_w(x, 0) = p_w^0(x) \qquad \forall x \in \Omega, \tag{2.10}$$

$$p_w(x, t) = p_{w_D}(x, t), s_n(x, t) = s_{n_D}(x, t) \qquad \forall x \in \Gamma_D, \tag{2.11}$$

$$v_\alpha \cdot n = J_\alpha(x, t), J_t = \sum_{\alpha \in \{w,n\}} J_\alpha \qquad \forall x \in \Gamma_N. \tag{2.12}$$

Here, J_α, $\alpha \in \{w, n\}$ is the inflow. In order to make p_w uniquely determined the Dirichlet boundary Γ_D should be of positive measure.

3 Discretization

Let $\mathcal{T}_h = \{E\}$ be a family of non-degenerate, quasi-uniform, possibly non-conforming partitions of Ω consisting of N_h elements (quadrilaterals or triangles in 2d, tetrahedrons or hexahedrons in 3d) of maximum diameter h. Let Γ^h be the union of the open sets that coincide with internal interfaces of elements of $\mathcal{T}_h$. Dirichlet and Neumann boundary interfaces are collected in the set Γ^h_D and Γ^h_N. Let e denote an interface in Γ^h shared by two elements E_- and E_+ of $\mathcal{T}_h$; we associate with e a unit normal vector n_e directed from E_- to E_+. We also denote by $|e|$ the measure of e. The discontinuous finite element space is $\mathcal{D}_r(\mathcal{T}_h) = \{v \in \mathbb{L}^2(\Omega) : v_{|E} \in \mathcal{P}_r(E) \ \ \forall E \in \mathcal{T}_h\}$, where $\mathcal{P}_r(E)$ denotes $\mathbb{Q}_r$ (resp. $\mathbb{P}_r$) the space of polynomial functions of degree at most $r \geq 1$ on E (resp. the space of polynomial functions of total degree $r \geq 1$ on E). We approximate the pressure and the saturation by discontinuous polynomials of total degrees r_p and r_s respectively.
For any function $q \in \mathcal{D}_r(\mathcal{T}_h)$, we define the jump operator $[\![\cdot]\!]$ and the average operator $\{\cdot\}$ over the interface e:

$\forall e \in \Gamma^h, \quad [\![q]\!] := q_{E_-} - q_{E_+}, \quad \{q\} := \frac{1}{2}q_{E_-} + \frac{1}{2}q_{E_+},$

$\forall e \in \partial\Omega, \quad [\![q]\!] := q_{E_-}, \quad \{q\} := q_{E_-}.$

In order to treat the strong heterogeneity of the permeability tensor, we follow [16] and introduce a weighted average operator $\{\cdot\}_\omega$:

$\forall e \in \Gamma^h, \ \ \{q\}_\omega = \omega_{E_-}q_{E_-} + \omega_{E_+}q_{E_+},$

$\forall e \in \partial\Omega, \ \ \{q\}_\omega = q_{E_-}.$

The weights are $\omega_{E_-} = \frac{\delta_K^{E_+}}{\delta_K^{E_+}+\delta_K^{E_-}}, \quad \omega_{E_+} = \frac{\delta_K^{E_-}}{\delta_K^{E_+}+\delta_K^{E_-}}$ with $\delta_K^{E_-} = n_e^T K_{E_-} n_e$ and $\delta_K^{E_+} = n_e^T K_{E_+} n_e$. Here, K_{E_-} and K_{E_+} are the permeability tensors for the elements E_- and E_+.

3.1 Semi discretization in space

The derivation of the semi-discrete DG formulation is standard (see [8], [16], [18]). First, we multiply each equation of (2.9) by a test function and integrate over each element, then we apply Green formula to obtain the semi-discrete weak DG formulation. Hence, the aforementioned formulation consists in finding the continuous in time approximations $p_{w,h}(\cdot,t) \in \mathcal{D}_{r_p}(\mathcal{T}_h)$, $s_{n,h}(\cdot,t) \in \mathcal{D}_{r_s}(\mathcal{T}_h)$ such that:

$$\mathcal{B}_h(p_{w,h},\varphi;s_{n,h}) = l_h(\varphi) \qquad \forall\varphi \in \mathcal{D}_{r_p}(\mathcal{T}_h), \forall t \in \mathcal{J}, \tag{3.1}$$
$$(\Phi\partial_t s_{n,h},\psi) + c_h(p_{w,h},\psi;s_{n,h}) + d_h(s_{n,h},\psi) = r_h(\psi) \qquad \forall\psi \in \mathcal{D}_{r_s}(\mathcal{T}_h), \forall t \in \mathcal{J}. \tag{3.2}$$

The bilinear form $\mathcal{B}_h$ in the total fluid conservation equation (3.1) is expressed as:

$$\mathcal{B}_h(p_{w,h},\varphi;s_{n,h}) = \mathcal{B}_{bulk,h} + \mathcal{B}_{cons,h} + \mathcal{B}_{sym,h} + \mathcal{B}_{stab,h}. \tag{3.3}$$

The first term $\mathcal{B}_{bulk,h}$ of (3.3) is the volume contribution:

$$\mathcal{B}_{bulk,h} := \mathcal{B}_{bulk,h}(p_{w,h},\varphi;s_{n,h}) = \sum_{E\in\mathcal{T}_h}\int_E(\lambda_t K\nabla p_{w,h} + \lambda_n K\nabla p_{c,h} - (\rho_n\lambda_n + \rho_w\lambda_w)Kg)\cdot\nabla\varphi. \tag{3.4}$$

The second term $\mathcal{B}_{cons,h}$, is the consistency term:

$$\begin{aligned}\mathcal{B}_{cons,h} := \mathcal{B}_{cons,h}(p_{w,h},\varphi;s_{n,h}) = &-\sum_{e\in\Gamma^h\cup\Gamma^h_D}\int_e\{\lambda_t K\nabla p_{w,h}\}_\omega[\![\varphi]\!]\\ &-\sum_{e\in\Gamma^h\cup\Gamma^h_D}\int_e\{\lambda_n K\nabla p_{c,h}\}_\omega[\![\varphi]\!]\\ &+\sum_{e\in\Gamma^h\cup\Gamma^h_D}\int_e\{(\rho_n\lambda_n + \rho_w\lambda_w)Kg)\}_\omega[\![\varphi]\!].\end{aligned} \tag{3.5}$$

The third term $\mathcal{B}_{sym,h}$, is the symmetry term. Depending on the choice of ϵ we get different DG methods ($\epsilon = -1$ SIPG, $\epsilon = 1$ NIPG, $\epsilon = 0$ IIPG):

$$\begin{aligned}\mathcal{B}_{sym,h} := \mathcal{B}_{sym,h}(p_{w,h}, \varphi; s_{n,h}) =& \epsilon \sum_{e\in\Gamma^h\cup\Gamma^h_D} \int_e \{\lambda_t K \nabla\varphi\}_\omega \cdot n_e [\![p_{w,h}]\!] \\ &+ \epsilon \sum_{e\in\Gamma^h\cup\Gamma^h_D} \int_e \{\lambda_n K \nabla\varphi\}_\omega \cdot n_e [\![s_{n,h}]\!].\end{aligned} \tag{3.6}$$

The last term $\mathcal{B}_{stab,h}$ is the stability term:

$$\mathcal{B}_{stab,h} := \mathcal{B}_{stab,h}(p_{w,h}, \varphi) = \sum_{e\in\Gamma^h\cup\Gamma^h_D} \gamma^p_e \int_e [\![p_{w,h}]\!][\![\varphi]\!]. \tag{3.7}$$

The right hand side of the total fluid conservation equation (3.1) is a linear form including the Neumann and Dirichlet boundary conditions and the source terms.

$$\begin{aligned}l_h(\varphi) =& \int_\Omega (q_w + q_n)\varphi - \sum_{e\in\Gamma_N} \int_e J_t \varphi + \epsilon \sum_{e\in\Gamma^h_D} \int_e \lambda_t K \nabla\varphi \cdot n_e p_D \\ &+ \epsilon \sum_{e\in\Gamma^h_D} \int_e \lambda_n K \nabla\varphi \cdot n_e s_D + l_{stab}, \qquad \forall \varphi \in \mathcal{D}_{r_p}(\mathcal{T}_h).\end{aligned} \tag{3.8}$$

Here, $l_{stab}(\varphi)$ is the stability term for the linear form:

$$l_{stab}(\varphi) = \sum_{e\in\Gamma^h_D} \gamma^p_e \int_e p_D \varphi. \tag{3.9}$$

Equation (3.2) is the discrete weak formulation of the nonwetting-phase conservation equation where the convection term $-\nabla\cdot(\lambda_n K(\nabla p_w - \rho_n g))$ might be approximated by an upwind discretization technique.

$$\begin{aligned}c_h(p_{w,h}, \psi; s_{n,h}) =& \sum_{E\in\mathcal{T}_h} \int_E (K\lambda_n(\nabla p_{w,h} - \rho_n g)) \cdot \nabla\psi - \sum_{e\in\Gamma^h\cup\Gamma^h_D} \int_e \{K\lambda^\#_n \nabla p_{w,h}\}_\omega \cdot n_e [\![\psi]\!] \\ &+ \sum_{e\in\Gamma^h\cup\Gamma^h_D} \int_e \{\rho_n K \lambda^\#_n g\}_\omega \cdot n_e [\![\psi]\!] + \epsilon \sum_{e\in\Gamma^h\cup\Gamma^h_D} \int_e \{K\lambda^\#_n \nabla\psi\}_\omega \cdot n_e [\![p_{w,h}]\!],\end{aligned} \tag{3.10}$$

where $\lambda^\#_n = (1-\varrho)\lambda_{n,E} + \varrho\lambda^\uparrow_n$ and $\lambda^\uparrow_n$ is the upwind mobility:

$$\forall e \in \partial E_- \cap \partial E_+, \quad \lambda^\uparrow_n = \begin{cases} \lambda_{n,E_-} & \text{if } -K(\nabla p_w + \nabla p_c - \rho_n g)\cdot n \geq 0, \\ \lambda_{n,E_+} & \text{else.}\end{cases}$$

Hence depending on the value of $\varrho \in \{0,1\}$, we might use central differencing or upwinding of the mobility for internal interfaces.
The diffusion term $-\nabla\cdot(\lambda_n K \nabla p_c)$ is discretized by a bilinear form similar to that of (3.3).

$$\begin{aligned}d_h(s_{n,h}, \psi) =& \sum_{E\in\mathcal{T}_h} \int_E \lambda_n K \nabla p_{c,h} \cdot \nabla\psi - \sum_{e\in\Gamma^h\cup\Gamma^h_D} \int_e \{\lambda_n K \nabla p_{c,h}\}_\omega \cdot n_e [\![\psi]\!] \\ &+ \epsilon \sum_{e\in\Gamma^h\cup\Gamma^h_D} \int_e \{\lambda_n K \nabla\psi\}_\omega \cdot n_e [\![s_{n,h}]\!] + \sum_{e\in\Gamma^h\cup\Gamma^h_D} \gamma^s_e \int_e [\![s_{n,h}]\!][\![\psi]\!].\end{aligned} \tag{3.11}$$

The right hand side r_h includes the Neumann and Dirichlet boundary condition and the nonwetting source term.

$$\begin{aligned} r_h(\psi) = \int_\Omega q_n \psi - \sum_{e \in \Gamma_N} \int_e J_n \psi + \epsilon \sum_{e \in \Gamma_D^h} \int_e \lambda_n K \nabla \psi \cdot n_e p_D \\ + \epsilon \sum_{e \in \Gamma_D^h} \int_e \lambda_n K \nabla \psi \cdot n_e p_c(s_D) + \sum_{e \in \Gamma_D^h} \gamma_e^s \int_e s_D \psi, \quad \forall \psi \in \mathcal{D}_{r_s}(\mathcal{T}_h). \end{aligned} \tag{3.12}$$

Remark 3.1 *The penalty terms γ_e^p and γ_e^s are discrete positive functions that take constant values on the interfaces. In order to ensure stability and convergence of the DG method, γ_e^p and γ_e^s must be chosen properly. This choice is especially crucial for strongly heterogeneous problems where parameters such as permeability, porosity, entry pressures can vary strongly, hence triggering a strong effect on the solution behavior. Following [8], we use in this work, unless specified otherwise, the penalty formulation as below.*

$$\gamma_e^p = C_p \frac{r_p(r_p + d - 1) \mid e \mid}{min(\mid E_- \mid, \mid E_+ \mid)}, \; C_p \geq 0 \tag{3.13}$$

and

$$\gamma_e^s = C_s \frac{r_s(r_s + d - 1) \mid e \mid}{min(\mid E_- \mid, \mid E_+ \mid)}, \; C_s \geq 0. \tag{3.14}$$

In the context of higher order discretization of large scale complex multiphase flow, the choice of proper basis functions is decisive for the computational efficiency and accuracy of the solver. In the sequel, we present the families of modal and nodal basis functions.

3.1.1 Modal basis The modal basis functions are sets of orthogonal polynomials w.r.t an appropriate inner product. They are also designed to have desirable properties such as hierachism, that is to say the basis for a given polynomial degree r includes the bases for polynomials degrees less than r. The use of hierarchical basis is essential for the prospect of higher order methods and local polynomial order adaptivity. The approximate solution $s_h^E(x,t)$ on each element E can be expressed as:

$$s_h^E(x,t) = \sum_{j=1}^{N_{loc}} \hat{s}_j^E(t) \psi_j(x), \;\; \forall E \in \mathcal{T}_h, \tag{3.15}$$

where the term $\{\hat{s}_j(t)\}_{j=1,\dots,N_{loc}}$ denotes the time dependent modal dofs and $\psi_j(x)$ is a d-dimensional polynomial basis. In the case of piecewise polynomials of total degree at most r, the local dimension N_{loc} is:

$$N_{loc} = \#\mathbb{P}_r = \frac{(r+d)!}{r!d!}. \tag{3.16}$$

In the case of piecewise polynomial of degree at most r in each variable the local dimension is:

$$N_{loc} = \#\mathbb{Q}_r = (r+1)^d. \tag{3.17}$$

A classical choice to generate modal basis functions $\psi_j(x)$ is to choose:

$$\psi_j(x) = P_{j+1}(x) - P_j(x), \quad j = 1, \dots, N_{loc}. \tag{3.18}$$

where P_j is the Legendre polynomial of degree j. It is also possible to use a Gramm-Schmidt procedure with the usual inner product to build an orthonormal basis from an initial monomial basis (see, e.g. [20]).

3.1.2 Nodal basis The nodal approach is based on Lagrange polynomials with roots at a set of nodal points. Therefore, the local approximation is:

$$s_h^E(x,t) = \sum_{i=1}^{N_{loc}} \tilde{s}^E(x_i,t) l_i^E(x), \tag{3.19}$$

where $l_i^E(x)$ is the d-dimensional Lagrange polynomial based on the nodal set $\{x_i\}_{i=1,\dots,N_{loc}}$.
Following Heasthaven [17], it is possible to switch from modal to nodal and vice-versa.

$$\tilde{s} = V\hat{s}, \qquad V^\top l(x) = \psi(x), \qquad V_{i,j} = \psi_j(x_i), \tag{3.20}$$

where V is the Vandermonde matrix containing the evaluation of modal polynomials at the interpolation points. This transformation allows to evaluate efficiently higher dimensional Lagrange polynomials $l_i(x)$.

3.2 Fully coupled/Fully implicit DG scheme

The time interval $[0,T]$ is divided into N intervals $\Delta t_i = t_{i+1} - t_i$ as $0 = t_0 \leq t_1 \leq \cdots \leq t_{N-1} \leq t_N = T$. Let p_w^i and s_n^i be the numerical solutions at time t^i. We also denote $\lambda_\alpha^i = \lambda_\alpha(s_n^i)$, $p_c^i = p_c(s_n^i)$. The approximation $s_{n,h}^0$ is chosen as the $\mathbb{L}^2$ projection of the saturation $s_n(0)$. For the sake of simplicity and easier reading, we apply a first order Adams-Moulton (Backward Euler) time discretization and Interior Penalty DG for space discretization to the semi-discrete system (3.1) - (3.2):

$$\mathcal{B}_h(p_{w,h}^{i+1}, \varphi; s_{n,h}^{i+1}) = l_h(\varphi), \qquad \forall \varphi \in \mathcal{D}_{r_p}(\mathcal{T}_h), \tag{3.21}$$

$$(\Phi \frac{s_{n,h}^{i+1} - s_{n,h}^i}{\Delta t}, \psi) + c_h(p_{w,h}^{i+1}, \psi; s_{n,h}^{i+1}) + d_h(s_{n,h}^{i+1}, \psi) = r_h(\psi), \qquad \forall \psi \in \mathcal{D}_{r_s}(\mathcal{T}_h), \tag{3.22}$$

$$(s_{n,h}^0, \zeta) = (s_n^0, \zeta), \qquad \forall \zeta \in \mathcal{D}_{r_s}(\mathcal{T}_h). \tag{3.23}$$

(3.21)-(3.23) leads to a large, nonlinear system of algebraic equations written in the form:

$$G(\bar{p}_{w,h}^{i+1}, \bar{s}_{n,h}^{i+1}) = \begin{pmatrix} G^{r_p}(\bar{p}_{w,h}^{i+1}, \bar{s}_{n,h}^{i+1}) \\ G^{r_s}(\bar{p}_{w,h}^{i+1}, \bar{s}_{n,h}^{i+1}) \end{pmatrix} = 0, \tag{3.24}$$

where $\bar{p}_{w,h}^{i+1} = (p_E)_E$ and $\bar{s}_{w,h}^{i+1} = (s_E)_E$ are vectors of unknowns for $p_{w,h}^{i+1}$ and $s_{n,h}^{i+1}$.
The system is solved by using a Newton-Raphson method.

$$\begin{aligned} J_G(\bar{p}_w^{i+1,r}, \bar{s}_n^{i+1,r})\delta^{r+1} &= -G(\bar{p}_w^{i+1,r}, \bar{s}_n^{i+1,r}), & (3.25) \\ (\bar{p}_w^{i+1,r+1}, \bar{s}_n^{i+1,r+1}) &= \delta^{r+1} + (\bar{p}_w^{i+1,r}, \bar{s}_n^{i+1,r}). & (3.26) \end{aligned}$$

Here, r denotes the rth Newton iterate and for a coupled system such as (3.25), the Jacobian J_G is:

$$J_G = \begin{pmatrix} J_{pp} & J_{ps} \\ J_{sp} & J_{ss} \end{pmatrix} = \begin{pmatrix} \frac{\partial G^{r_p}}{\partial p} & \frac{\partial G^{r_p}}{\partial s} \\ \frac{\partial G^{r_s}}{\partial p} & \frac{\partial G^{r_s}}{\partial s} \end{pmatrix}.$$

4 Adaptivity strategy

For the considered DG discretization of porous media two-phase flow problem, derivation of error estimates based on rigorous a-posteriori error estimates is out of the scope of this paper. Hence, we use heuristic indicators which depend on the local gradient of the nonwetting-phase-saturation s_n measured in the $\mathbb{L}^2$ norm. We define on each element E of the mesh, the indicator η_E^i at time step i, such that:

$$\eta_E^i = \|\nabla s_n^i\|_{L^2(E)}, \qquad \forall E \in \mathcal{T}_h. \tag{4.1}$$

Each element whose indicator η_E^i is greater than a treshold value $\eta_{Tol} \geq 0$ is refined.

5 Implementation using `Dune-Fem`

This section is dedicated to an overview of the implementation of the DG two-phase flow simulator based on `Dune-Fem`. For a more in-depth description of the `Dune-Fem` interface, we refer to [12].

5.1 Library requirements

`Dune-twophaseDG` needs the `Dune` core modules `Dune-Common`, `Dune-Grid`, `Dune-Localfunctions`, `Dune-Istl` at version 2.3 (or later) and the `Dune-Fem` module at version 1.4 (or later). For performing h-adaptivity, one has to use adaptive grids such as `Alugrid_Cube` or `Alugrid_Simplex` from the library `Dune-Alugrid`, non-adaptive grids such as `YaspGrid` or `SGrid` do not provide local adaptivity.

5.2 Structure and code description

We describe the structure of the directory of `Dune-twophaseDG` in terms of subdirectories, header files and executable files. The following subdirectories are within the module:

- *CMake*: configuration options for building the module while using Cmake.
- *doc*: doxygen documentation.
- *dune*: header files.
- *src*: source files for the numerical examples.

5.2.1 The directory *dune*

The directory *dune* contains different subdirectories:

- *algorithm*: contains the header files `algorithm.hh`, `femscheme.hh`, `phaseflowscheme.hh`, `probleminterface.hh` and `temporalprobleminterface.hh`.
- *estimator*: contains the header file `estimator.hh` providing a heuristic estimator for the numerical solution.
- *models*: contains the header file `phaseflowmodel.hh`.
- *operator*: contains the header files `operator.hh` providing the classes which build the discrete stiffness matrix and the right hand side. The header file `newtoninvop.hh` provides the Newton inverse operator.

5.2.2 The directory *examples*

The directory *examples* contains the *lenspb* folder holding the infiltration problem header files `lenspbbndmodel.hh`, `lenspbinitialdata.hh`, `lenspbmodel.hh` and `lenspbphysicalparmodel.hh`.

5.2.3 The directory *src*

The *src* directory contains the source files for the different numerical modules:

- *3dlens*: 3d infiltration problem with gravity forces and capillarity effects,
- *2dlens*: 2d infiltration problem with gravity forces and capillarity effects.

In each test, we have a source file for the main program (i.e. `3dlens.cc`, `2dlens.cc`).

5.3 Features of the implementation

The implementation of the discrete formulation (3.21)-(3.23) is realized in a similar fashion as most of the tutorial examples from the Dune-Fem-howto. Starting from the included heat DG problem, we extend it to a system of two equations and two unknowns and we add the non-linear formulation. First, we describe the PDE by a class phaseflowmodel, which serve as an interface for general test cases. This is also the interface used to implement the operator. In the phaseflowmodel (see Code 1), the methods wetting_Pressure_DiffusiveFlux() for the wetting pressure flux, capillary_Pressure_DiffusiveFlux() for the capillary pressure flux and gravity_Flux() for the gravity flux will return the terms multiplied with ∇v. The source() method will return the parts of the operator multiplied with v. In order to handle the non-linearity, we also add the methods linSource() (resp. linWetting_Pressure_DiffusiveFlux(), linCapillary_Pressure_DiffusiveFlux() and linGravity_Flux()) returning the linearization of the source term (resp. flux terms). The DNAPL infiltration test case has its own model lenspbmodel which derives from the phaseflowmodel. The initial and boundary data are specified respectively in lenspbinitialdata.hh and lenspbbndmodel.hh. The lenspbphysicalparmodel class specifies the different physical properties of the lens problem such as the mobility and the capillary pressure function formulations.

```
template< class FunctionSpace, class GridPart, class PhysParModel  >
struct PhaseFlowModel
{
    template< class Entity, class Point >
  void source ( const Entity &entity,
                const Point &x,
                const RangeType &value,
                RangeType &flux ) const;

  template< class Entity, class Point >
  void linSource ( const RangeType& valueUn,
                   const Entity &entity,
                   const Point &x,
                   const RangeType &value,
                   RangeType &flux ) const;


  //! return the wetting pressure flux
  template< class Entity, class Point >
  void wetting_Pressure_DiffusiveFlux ( const Entity &entity,
                                        const Point &x,
                                        const RangeType &value,
                                        const JacobianRangeType &gradient,
                                        JacobianRangeType &flux,
                                        const double lambda_n_upw=1,
                                        const bool &useupw=false,
                                        const bool &buildRhs=false ) const;

  //! return the linearized wetting pressure flux
  template< class Entity, class Point >
  void linWetting_Pressure_DiffusiveFlux ( const RangeType &valueUn,
                                           const JacobianRangeType &gradientUn,
                                           const Entity &entity,
                                           const Point &x,
                                           const RangeType &value,
                                           const JacobianRangeType &gradient,
                                           JacobianRangeType &flux,
                                           const double lambda_n_upw = 1,
                                           const double gradnonwetmob_upw = 1,
                                           const bool useupw=false ) const;
}
```

Code 1: Excerpt from phaseflowmodel.hh.

The assembly process of the operator and the right hand side is done in the file operator.hh. The class FlowOperator is derived from the Dune::Operator class. This is why we need to override the operator() method. In order to build the jacobian, we introduce a class DifferentiableFlowOperator (see Code 2) which derives from the FlowOperator and from the interface class DifferentiableOperator:

```
template< class JacobianOperator, class Model >
struct DifferentiableFlowOperator
  : public FlowOperator< typename JacobianOperator::DomainFunctionType, Model >,
    public Dune::Fem::DifferentiableOperator< JacobianOperator >
```

Code 2: DifferentiableFlowOperator.

In order to build the operator, we iterate over the intersections of the elements. For each intersection, we evaluate the local functions on the elements on both sides of the intersection by using a `FaceQuadratureType::INSIDE` for the element and `FaceQuadratureType::OUTSIDE` for the neighboring element. Code 3 shows the assembly process of the operator where the method `volumetricPart()` (line 24) computes the local contribution from each element and `internal_Bnd_terms()` (line 34) computes local contribution from interfaces and boundaries.

```
 1 template< class DiscreteFunction, class Model >
 2 void FlowOperator< DiscreteFunction, Model >
 3 ::operator() ( const DiscreteFunctionType &u, DiscreteFunctionType &w ) const
 4 {
 5   // clear destination
 6   w.clear();
 7   // get discrete function space
 8   const DiscreteFunctionSpaceType &dfSpace = w.space();
 9 
10   // iterate over grid
11   const IteratorType end = dfSpace.end();
12   for( IteratorType it = dfSpace.begin(); it != end; ++it )
13   {
14     // get entity (here element)
15     const EntityType &entity = *it;
16     // get elements geometry
17     const GeometryType &geometry = entity.geometry();
18     // get local representation of the discrete functions
19     const LocalFunctionType uLocal = u.localFunction( entity );
20     LocalFunctionType wLocal = w.localFunction( entity );
21     // obtain quadrature order
22     const int quadOrder = uLocal.order() + wLocal.order();
23     //Computing local contribution from elements
24     volumetricPart(entity, quadOrder, geometry, uLocal, wLocal);
25 
26     if ( ! dfSpace.continuous() )
27     {
28       const IntersectionIteratorType iitend = dfSpace.gridPart().iend( entity );
29       // looping over intersections
30       for( IntersectionIteratorType iit = dfSpace.gridPart().ibegin( entity ); iit != iitend; ++iit
         )
31       {
32         const IntersectionType &intersection = *iit;
33         //Computing local contribution from interfaces and boundaries
34         internal_Bnd_terms(entity, quadOrder, geometry, intersection, dfSpace, uLocal, u, wLocal);
35       }
36     }
37   }
38   // communicate data (in parallel runs)
39   w.communicate();
40 }
```

Code 3: Operator building.

The selection of the type of discrete function space is done in the `femscheme` class. The discrete function space depends on the `GridPartType` and the `FunctionSpace` (see Code 4). It allows to choose the polynomial order by setting the parameter `POLORDER`, hence permitting the use of higher order polynomials without any further changes in the code. For the sake of simplicity and usability, the code only supports the case $r_p = r_s =$ `POLORDER`.

```
 //! choose type of discrete function space and the polynomial order POLORDER
#if USE_LAG_DG
  typedef Dune::Fem::LagrangeDiscontinuousGalerkinSpace< FunctionSpaceType, GridPartType, POLORDER
     > DiscreteFunctionSpaceType;
#else
  typedef Dune::Fem::DiscontinuousGalerkinSpace< FunctionSpaceType, GridPartType, POLORDER >
     DiscreteFunctionSpaceType;
#endif
```

Code 4: Type of discrete function space and polynomial order.

In order to achieve an adaptive scheme, we implement an `estimator` class which supports a method `mark()` (see Code 5) to mark the elements for the next refinement step. Our marking strategy consist in looping over the mesh and selecting for refinement all elements where the $\mathbb{L}^2$ norm of the saturation gradient is larger than a certain tolerance η_{Tol}. The value of the η_{Tol} can be specified in the parameter file with the variable `phaseflow.tolerance`.

```
   //! mark all elements due to given tolerance
  bool mark ( const double tolerance ) const
  {
    int marked = 0;
    // loop over all elements
    const IteratorType end = dfSpace_.end();
    for( IteratorType it = dfSpace_.begin(); it != end; ++it )
      {
        const ElementType &entity = *it;


        const Dune::ReferenceElement< double, dimension > &refElement
        = Dune::ReferenceElements< double, dimension >::general( entity.type() );
        RangeType val;
        // evaluate the phase field at the barycentre (note
        // refElement.position(0,0) is the barycentre in local coordinates)
        double markVal;
        JacobianRangeType grad;
        uh_.localFunction(entity).jacobian(refElement.position(0,0),grad);
        markVal = grad[1].two_norm();


        if( markVal > tolerance )
        {
          // make sure grid is not overly refined...
          // maxLevel_ is the maximum level of refinement allowed
          if (entity.level() < maxLevel_)
          {
            // mark entity for refinement
            grid_.mark( 1, entity );
            // grid was marked
            marked = 1;
          }
        }
       else
       {
          // mark for coarsening
          grid_.mark( -1, entity );
       }
      }
    // get global max
    marked = grid_.comm().max( marked );
    return bool(marked);
  }
```

Code 5: Excerpt from `estimator.hh`.

In the main function (see Code 6), we first initialize `MPI`, then read the parameters and construct the grid based on the grid implementation provided in `CMakeLists.txt`. After initializing the grid, we get an instance of the class `Algorithm` containing the algorithm (line 26). After that, the function compute is executed in line 30.

```
1  int main ( int argc, char **argv )
2    try
3      {
4        // initialize MPI, if necessary
5        Dune::Fem::MPIManager::initialize( argc, argv );
6        // append overloaded parameters from the command line
7        Dune::Fem::Parameter::append( argc, argv );
8        // append possible given parameter files
9        for( int i = 1; i < argc; ++i )
10        Dune::Fem::Parameter::append( argv[ i ] );
11       // append default parameter file
12       Dune::Fem::Parameter::append( "../data/parameter" );
13       // type of hierarchical grid
14       typedef Dune::GridSelector::GridType  HGridType ;
15       typedef Algorithm< HGridType > AlgorithmType;
16       // create grid from DGF file
17       const std::string gridkey = Dune::Fem::IOInterface::defaultGridKey( HGridType::dimension );
18       const std::string gridfile = Dune::Fem::Parameter::getValue< std::string >( gridkey );
```

```
      // the method rank and size from MPIManager are static
      if( Dune::Fem::MPIManager::rank() == 0 )
       std::cout << "Loading macro grid: " << gridfile << std::endl;
      // construct macro using the DGF Parser
      Dune::GridPtr< HGridType > gridPtr( gridfile );
      HGridType& grid = *gridPtr ;

      AlgorithmType myalgorithm (grid);
      // Compute algorithm
      Dune::Timer computetimer;
      //Compute the algorithm
      myalgorithm.compute();
      const double compuTime = computetimer.elapsed();

      ...

      return 0;
    }
```

Code 6: main function of 2dlens.cc.

In the `compute()` method of the class `Algorithm` (Code 7), we initialize two model instances which are passed on to the `Scheme` class. Here, two elliptic operators are constructed and used to evolve the solution from one time level to the next. The `TimeProvider` class is used to handle time dependency.

```
     // create time provider
    Dune::Fem::GridTimeProvider< HGridType > timeProvider( grid_ );
    // we want to solve the problem on the leaf elements of the grid
    GridPartType gridPart(grid_);
    // type of the mathematical model used
    ProblemType problem( timeProvider ) ;
    // implicit model for left hand side
    ModelType implicitModel( problem, gridPart, true );
    // explicit model for right hand side
    ModelType explicitModel( problem, gridPart, false );
    // create scheme
    SchemeType scheme( gridPart, implicitModel, explicitModel );
    //! input/output tuple and setup datawritter
    IOTupleType ioTuple( &(scheme.solution())) ; // tuple with pointers
    DataOutputType dataOutput( grid_, ioTuple );

    const bool local_adapt = Dune::Fem::Parameter::getValue< bool >("phaseflow.local_adapt", false
     );
    const double endTime  = Dune::Fem::Parameter::getValue< double >( "phaseflow.endtime", 3.0 );
    const double dtreducefactor = Dune::Fem::Parameter::getValue< double >("phaseflow.
     reducetimestepfactor", 1 );
    double timeStep = Dune::Fem::Parameter::getValue< double >( "phaseflow.timestep", 0.00125 );
    double tolerance = Dune::Fem::Parameter::getValue< double >("phaseflow.tolerance", 0.5 );

    int step=1;
    timeStep *= pow(dtreducefactor,step);
    // initialize with fixed time step
    timeProvider.init( timeStep ) ;
    scheme.initialize();
    // write initial solve
    dataOutput.write( timeProvider );


    // time loop, increment with fixed time step
    for( ; timeProvider.time() < endTime; timeProvider.next( timeStep ) )
      {
        std::cout << "t=" << timeProvider.time() << std::endl;

        if (local_adapt)
         {
           // mark element for adaptation
           scheme.mark( tolerance );
           // adapt grid
           scheme.adapt();
           scheme.prepare();
           scheme.solve(true);
           scheme.postpro();
         }
  ...
      }
```

Code 7: Excerpt from `algorithm.hh`.

5.4 Input & Output files

5.4.1 Input files

We use parameter files (see Code 8) to set parameters for the simulation.

```
# GENERAL #######################
#--------


#### Parameters for output ######
phaseflow.computeEOC: 0
# prefix data files
fem.io.datafileprefix: 2dtwophase
# save every i-th step
fem.io.savestep: 40.0e00


# specify directory for data output (is created if not exists)
fem.prefix: ../Output2D/Deg3/LagBasis


# upwinding of advective term
phaseflow.with_upw: false


#local adaptivity
phaseflow.local_adapt: true


# tolerance for estimator
phaseflow.tolerance: 5


#number of level of refinement
phasefield.maxlevel:2


#DG penalty
phaseflow.penaltypress: 1e-2
phaseflow.penaltysat: 1e-3


#DG method NIPG=-1 SIPG=1 IIPG=0
phaseflow.DGeps: 1


#################################
```

Code 8: Excerpt from the parameter file.

The input files are read in by the compiled program. Thus values can be modified at runtime (see Code 9).

```
// append default parameter file
Dune::Fem::Parameter::append( "../data/parameter" );
```

Code 9: Loading of the input file.

The `Dune::Parameter` singleton parses the given parameter file line by line. Code 10 shows how to use `Dune::Parameter`. The method `getValue()` expects two parameters. The first one is a `std::string`. The last one is a default value that will be used if the value of the parameter is not provided in the input file. This last parameter is optional.

```
const bool local_adapt = Dune::Fem::Parameter::getValue< bool >("phaseflow.local_adapt", false );
const double endTime  = Dune::Fem::Parameter::getValue< double >( "phaseflow.endtime", 3.0 );
double timeStep = Dune::Fem::Parameter::getValue< double >( "phaseflow.timestep", 0.00125 );
```

Code 10: Example of the `getValue()` method utilisation.

The grid type can either be specified directly or obtained from the `GridSelector`. The type is then specified during the make or `CMake` procedure.

```
 add_definitions(
-DALUGRID_CUBE
-DPOLORDER=3
-DGRIDDIM=2
-DWORLDDIM=2
-DWANT_ISTL=0
-DUSE_LAG_DG=1
 )
 add_executable(2dlens 2dlens.cc)
 add_dune_alugrid_flags(2dlens)
```

For more in-depth information on the input files we refer to the DUNE documentation [12].

5.4.2 Output files

The output is handled by the DataOutput class (see Line 4, Code 11) and a tuple holding pointers to DiscreteFunction objects is passed as a parameter. The generated vtu files are exported into the directory specified in the parameter file by fem.prefix (see Line 14, Code 8).

```
1  // type of input/output
2  typedef Dune::tuple< DiscreteFunctionType* > IOTupleType;
3  // type of the data writer
4  typedef Dune::Fem::DataOutput< HGridType, IOTupleType > DataOutputType;
5  IOTupleType ioTuple( &(scheme.solution())) ; // tuple with pointers
6  DataOutputType dataOutput( grid, ioTuple, DataOutputParameters( step ) );
```

Code 11: Output handling.

6 Numerical simulations

In this section we present some numerical tests for the presented DG scheme. Unless specified otherwise, all test cases are implemented with either the SIPG or the IIPG method. In order to ensure second order accuracy, we employ a central differencing of the mobility for internal interfaces thus following a similar approach to that of Rivière et al. [15]. We do not use any kind of slope limiting or upwinding techniques. The linear solver used is GMRES and we do not use any preconditioner. The maximal polynomial order employed for the 2d problem is $r_p = r_s = 3$. Although it is possible to use higher polynomial order (quartics and quintics), the schemes become computationally expensive in terms of both storage and CPU time for practical use.

6.1 Test Case 1: A vertical DNAPL infiltration Flow over a low permeability lens

A container is filled with two kinds of sand and saturated with water with density $\rho_w = 1000\ Kg/m^3$ and viscosity $\mu_w = 1 \times 10^{-3}\ Kg/m\ s$. The DNAPL considered in the experiment is Tetrachloroethylene with density $\rho_n = 1460\ Kg/m^3$ and viscosity $\mu_n = 9 \times 10^{-4}\ Kg/m\ s$.

Brooks-Corey's constitutive relations are used for the capillary pressure and the relative permeabilities. Discretization of the system is performed by Interior Penalty DG methods with a fully implicit/fully coupled approach. All test cases in this section include gravitational forces and capillary pressure effects. We use ALUCubeGrid for the test cases, it implements the Dune GridInterface for 3d hexahedral meshes. The grids can be locally adapted (non-conforming) and used in parallel computations [1].

6.1.1 $2d$ infiltration problem We consider here a two-dimensional DNAPL infiltration problem with different sand types. The bottom of the reservoir is impermeable for both phases. Hydrostatic conditions for the pressure p_w and homogeneous Dirichlet conditions for the saturation s_n are prescribed at the left and right boundaries. A flux of $J_n = -5.137 \times 10^{-5}\ m\ s^{-1}$ of the DNAPL is infiltrated into the domain from the top. Detailed boundary conditions are specified in Table 2 and Figure 1. Initial conditions where the domain is fully saturated with water and hydrostatic pressure distribution are considered (i.e. $p_w^0 = (0.65 - y) \cdot 9810$, $s_n^0 = 0$). The initial mesh consists of 600 quadrilateral elements. We choose a time step of size $\Delta t = 5\ s$. The final time is $T = 2000\ s$. We consider a Newton solver tolerance $newtTol = 3 \times 10^{-7}$ and a linear solver tolerance $linabstol = 2.7 \times 10^{-7}$.

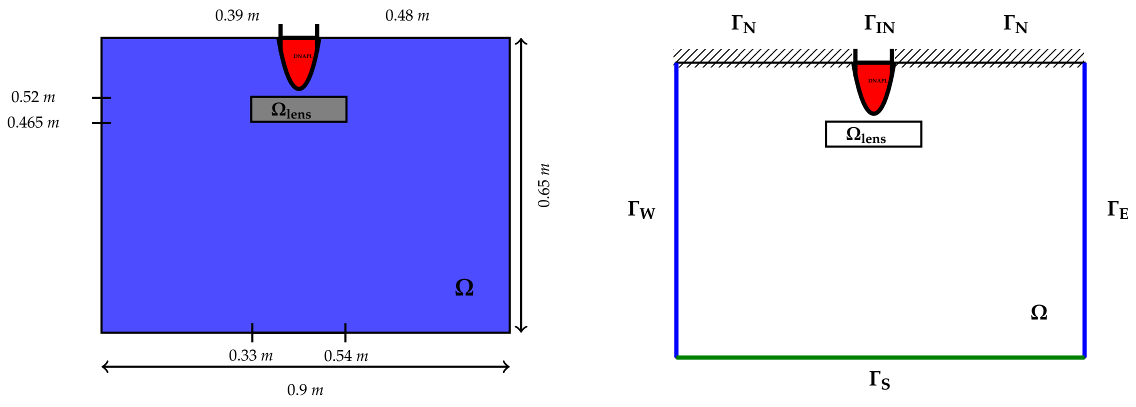


Figure 1: Geometry and boundary conditions for the DNAPL infiltration problem.

	Ω_{lens}	$\Omega \backslash \Omega_{lens}$
Φ [-]	0.39	0.40
k [m^2]	6.64×10^{-16}	6.64×10^{-11}
S_{wr} [-]	0.1	0.12
S_{nr} [-]	0.00	0.00
θ [-]	2.0	2.70
p_d [Pa]	5000	755

Table 1: Parameters.

Γ_{IN}	$J_n = -5.137 \times 10^{-5}$, $J_w = 0$
Γ_N	$J_n = 0.00$, $J_w = 0.00$
Γ_S	$J_w = 0$, $J_n = 0.00$
$\Gamma_E \cup \Gamma_W$	$p_w = (0.65 - y) \cdot 9810$, $s_n = 0$

Table 2: Boundary conditions.

Figure 2 and Figure 3 show the numerical results for the IIPG scheme with polynomial order $r_s = r_p = 3$ combined with first (resp. second) order Adams-Moulton method time discretization. We use here Lagrange DG space. The implementation of the Lagrange DG space is done by the mean of a Vandermonde matrix operating the transformation from spectral to physical space.

It took 520 s for the DNAPL to reach the lens and to spread out in the horizontal direction until reaching the edge of the lens. Afterwards the nonwetting front propagates down the sides of the lens. However as expected for advection dominated problems, we witness severe undershoots in the vicinity of the free boundary. The local h-adaptivity allows us to reduce those undershoots to small values. Figure 4 and Figure 5 show a comparison between the modal and Lagrange DG schemes. We witness smeared fronts for the orthonormal monomial basis unless we use small values of penalisation. The shape of the front for the Lagrange basis are less diffusive for large values of the penalisation parameter. Table 3 throws light upon the columns labels used in the numerical results. In Table 4, we provide details of the simulation including total computation times. As expected, the total computation time increases substantially with higher order polynomial degree. One can't help but notice that almost 80% (resp. 70%) of the total computing time is spent building the jacobian matrix for the DG/$\mathbb{Q}_3$ AM1 (resp. DG/$\mathbb{Q}_3$ AM2).

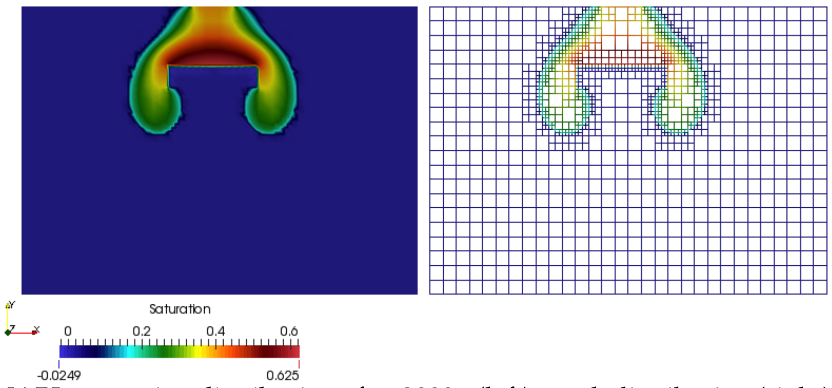


Figure 2: DNAPL saturation distribution after 2000 s (left), mesh distribution (right). Polynomial order $p = 3$.

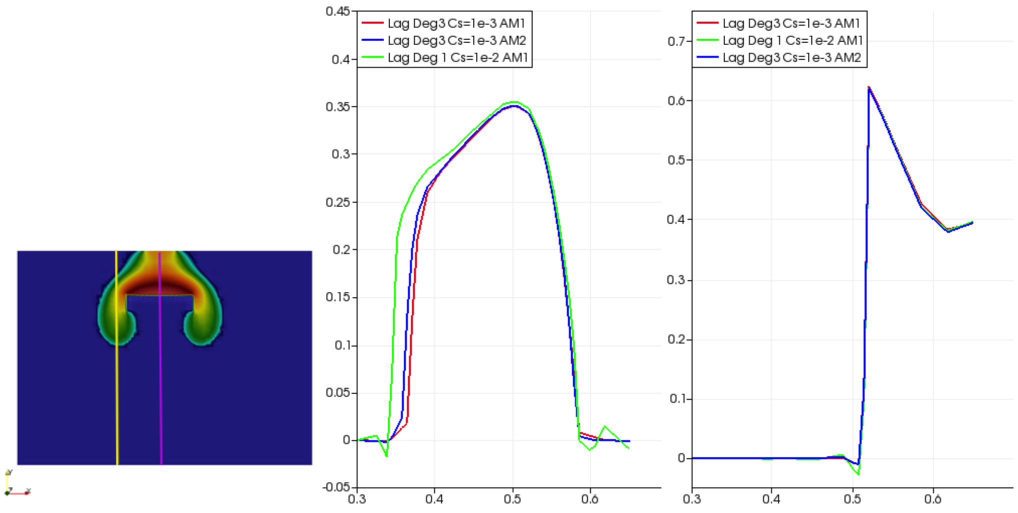


Figure 3: Comparison of non-wetting-phase saturation at T=2000 s. Center, profile along the line x=0.3 m (yellow vertical line of the left figure). Right, profile along the line x=0.45 m (violet vertical line of the left figure).

DG/$\mathbb{Q}_1$ AM1	Piecewise linear Lagrange DG combined with first order Adams-Moulton
DG/$\mathbb{Q}_3$ AM1	Piecewise cubic Lagrange DG combined with first order Adams-Moulton
DG/$\mathbb{Q}_3$ AM2	Piecewise cubic Lagrange DG combined with second order Adams-Moulton
Avg nb lin iter / Newton cycle	Average number of linear iterations per Newton cycle
Avg assem time / lin iter	Average time to assemble the Jacobian matrix per linear iteration
Avg inv time / lin iter	Average time to invert the Jacobian matrix per linear iteration

Table 3: Notation in result representation.

	DG/$\mathbb{Q}_1$ AM1	DG/$\mathbb{Q}_3$ AM1	DG/$\mathbb{Q}_3$ AM2
Avg nb lin iter / Newton cycle	486.869	310.574	517.365
Avg assem time / lin iter [sec]	0.75	9.78	9.25
Avg inv time / lin iter [sec]	0.21	2.67	4.03
Total cpu time [sec]	555.595	9669.63	10609.2

Table 4: Runtime overview for the 2d DNAPL infiltration problem.

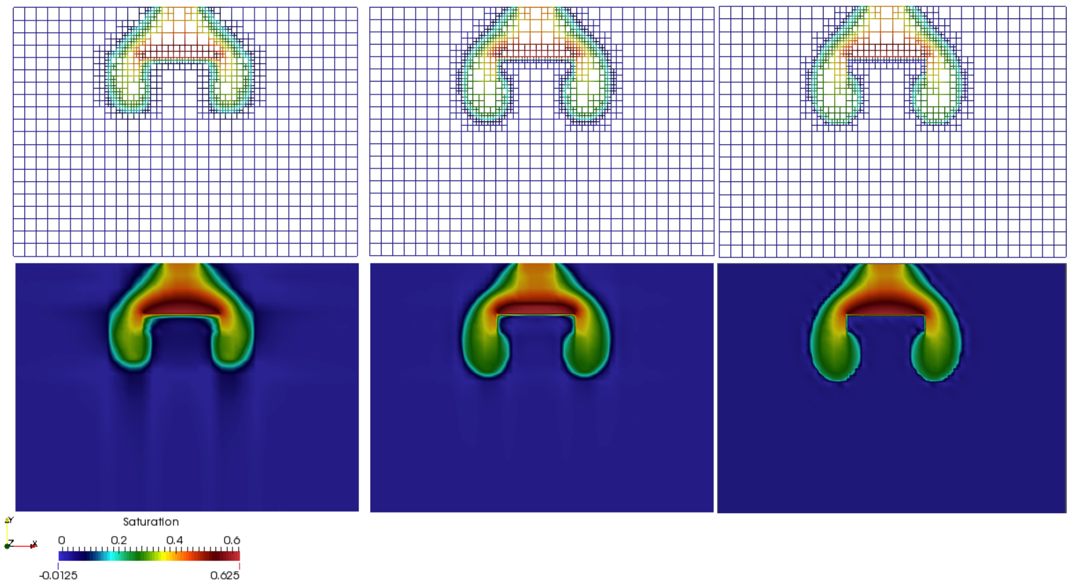


Figure 4: Contour plot of DNAPL saturation after 2000 s for modal orthonormal basis with $C_s = 1e-2$ (left column), modal orthonormal basis with $C_s = 1e-3$ (center column) and Lagrange basis with $C_s = 1e-2$ (right column). Polynomial order $p = 1$.

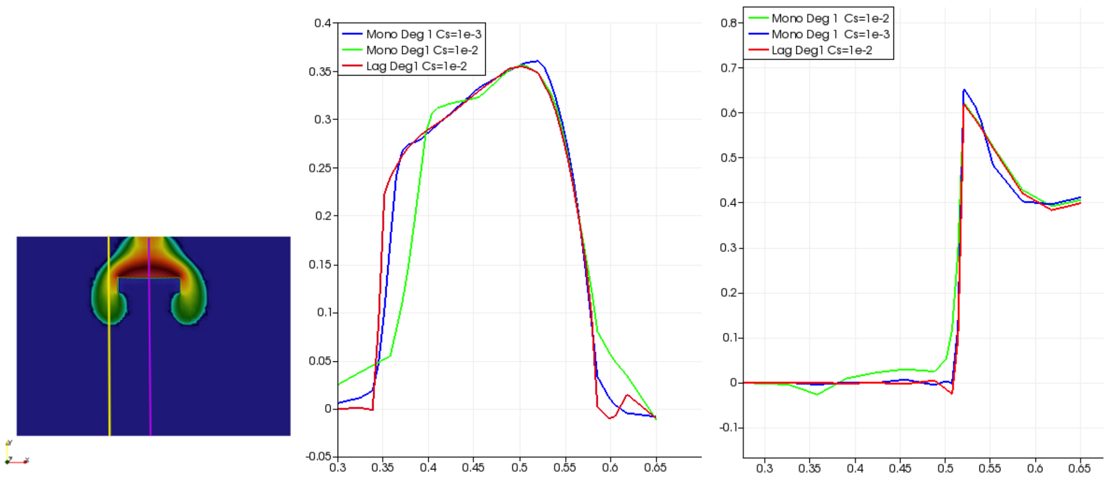
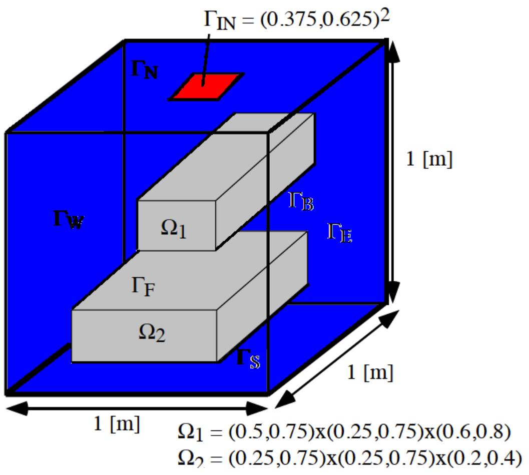


Figure 5: Comparison of non-wetting-phase saturation at T=2000 s. Center, profile along the line x=0.3 m (yellow vertical line of the left figure). Right, profile along the line x=0.45 m (violet vertical line of the left figure).

6.1.2 $3d$ **infiltration problem** In this section, we extend the previous results to the three-dimensional case. We also consider different sand types with different permeabilities and different entry pressures. The bottom of the reservoir is impermeable for both phases. Hydrostatic conditions for the pressure p_w and homogeneous Dirichlet conditions for the saturation s_n are prescribed at the lateral boundaries. A flux of $J_n = -1.712 \times 10^{-4}\ m\, s^{-1}$ of the DNAPL is infiltrated into the domain from the top. The initial ALUCubeGrid mesh consist of $10 \times 10 \times 10$ hexahedral elements and resolves the interfaces between regions with different permeabilities. 60 time steps of length $\Delta t = 60\ s$ are computed (final time $T = 3600\ s$). This grid is locally adapted (non-conforming).



$\Gamma_{IN} = (0.375,0.625)^2$
Γ_N
Γ_W
Γ_B
Γ_E
Γ_F
Γ_S
Ω_1
Ω_2
1 [m]
1 [m]
1 [m]
$\Omega_1 = (0.5,0.75)x(0.25,0.75)x(0.6,0.8)$
$\Omega_2 = (0.25,0.75)x(0.25,0.75)x(0.2,0.4)$

Figure 6: Geometry of the domain for the 3d DNAPL infiltration problem.

	Ω_1	Ω_2	$\Omega\backslash\Omega_1 \cap \Omega\backslash\Omega_2$
Φ [-]	0.39	0.39	0.40
k [m^2]	6.64×10^{-16}	6.64×10^{-15}	6.64×10^{-11}
S_{wr} [-]	0.1	0.1	0.12
S_{nr} [-]	0.00	0.00	0.00
θ [-]	2.0	2.0	2.70
p_d [Pa]	5000	5000	755

Figure 7: 3d problem parameters.

Figure 8 illustrates the evolution of the nonwetting saturation during the simulation. We show results at 3600 s of simulation time. As we increase the polynomial order, we notice undershoots in the vicinity of the front of the propagation and a sharp discontinuity in the solution at the lenses interfaces.

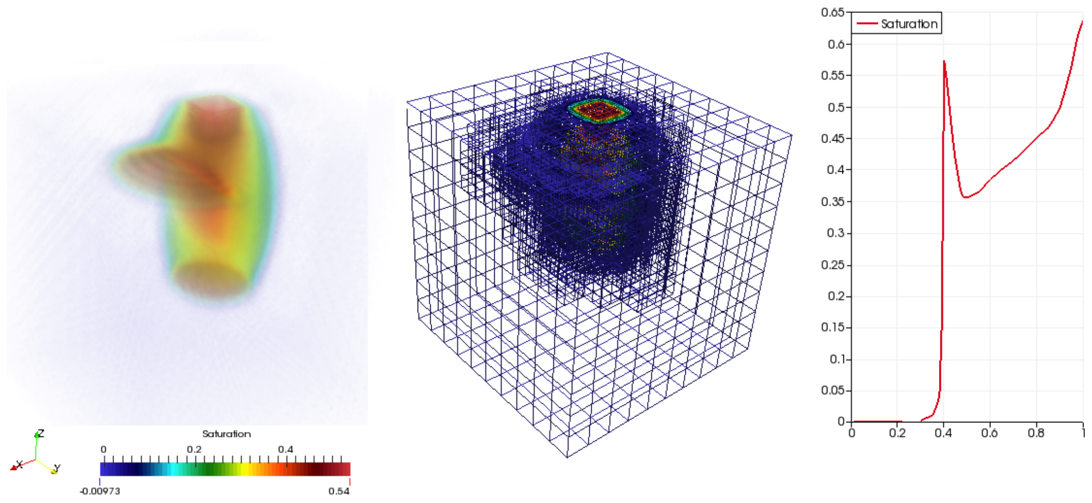


Figure 8: Contour plot of saturation distribution after 3600 s of DNAPL injection in a depth of 1 m (left column), mesh distribution (center column) and saturation profile along the line ((0.45,0.45,0);(0.45,0.45,1))(right column).

7 Conclusion & Outlook

In this work, we presented a discontinuous Galerkin scheme for incompressible, immiscible two-phase flow in strongly heterogeneous porous media with gravity forces and discontinuous capillary pressures. Higher-order polynomials up to piecewise cubics are implemented. The different test cases considered depict convection dominated problems such as DNAPL infiltration in an initially water saturated reservoir. The oscillations appearing in the the vicinity of the front of the propagation are reduced with the local mesh refinement. Undoubtedly this fully implicit DG requires further theoretical and numerical research. DG schemes such as Compact Discontinuous

Galerkin 2 (CDG2)[9], which are less sensitive to the penalty parameter value seem to be suitable candidates for our models. Derivation of more rigorous a posteriori estimates that can robustly estimate both temporal and spatial error are of dire interest for more efficient adaptive algorithms. Effective local slope limiters are also needed in order to avoid the undershoots and overshoots witnessed in the vicinity of the front of the propagation.

Apendix A

Building the library

We present here the main steps to create a local working installation of Dune-twophaseDG.

- Create a DuneWorkspace directory.

```
$ mkdir DuneWorkspace && cd DuneWorkspace
```

- Checkout the latest (stable) core modules from the Dune project homepage.

```
$ for MOD in common geometry grid localfunctions istl; do
$ git clone -b releases/2.4 https://gitlab.dune-project.org/core/dune-$MOD.git
$ done
```

- Checkout the latest (stable) version of Dune-Fem.

```
$ git clone -b releases/2.4 https://users.dune-project.org/repositories/projects/dune-fem.git
```

- Checkout Dune-Alugrid (required for the local grid adaptivity).

```
$ git clone -b releases/2.4  https://users.dune-project.org/repositories/projects/dune-alugrid.git
```

- Checkout Dune-twophaseDG.

```
$ git clone   https://gitlab.dune-project.org/birane.kane/dune-twophaseDG.git
```

Remark 7.1 *You can also download and unpack a Dune-twophaseDG tarball to a folder in your file system and extract the content of the tar files. Make sure that the extracted Dune-twophaseDG is in the DuneWorkspace directory.*

- Configure and compile the library by typing the following command in the DuneWorkspace directory.

```
$ cp dune-twophaseDG/scripts/opts/cmake.opts ./
$ ./dune-common/bin/dunecontrol --opts=cmake.opts all
```

Run of a test application

We assume in this section that the compilation of all required libraries has been completed in accordance with the description given in the previous section. The numerical model of `Dune-twophaseDG` are compiled in a build-folder (default: `build-cmake`) and tested in the test subfolder. For example, to run the 3d lens problem:

```
$ cd build-cmake/src/test/lenspb/3dlens
$ make 3dlens
$ ./3dlens -parameterFile ./../data/parameter3d
```

The parameter file specifies that all important parameters (like first time-step size, end of simulation and location of the grid file) can be found in a text file in the data directory with the name param*. The simulation starts and produces some .vtu output files and also a .pvd file in the folder `build-cmake/src/test/lenspb/Output3D`.

Remark 7.2 *All the test cases presented in this work can be executed with the following command.*

```
$ cd build-cmake/src/test/lenspb
$ source ../../../../scripts/allnumtest.sh
```

Acknowledgements

The author would like to acknowledge Dr. Bernd Flemisch and Dr. Claus J. Heine for valuable suggestions. Furthermore, the author would like to thank the German Research Foundation (DFG) for financial support of the project within the Cluster of Excellence in Simulation Technology (EXC 310/2) at the University of Stuttgart.

References

[1] M. Alkämper, A. Dedner, R. Klöfkorn, and M. Nolte. The dune-alugrid module. *arXiv preprint arXiv:1407.6954*, 2014.

[2] D. N. Arnold. An interior penalty finite element method with discontinuous elements. *SIAM journal on numerical analysis*, 19(4):742–760, 1982.

[3] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous galerkin methods for elliptic problems. *SIAM journal on numerical analysis*, 39(5):1749–1779, 2002.

[4] G. A. Baker. Finite element methods for elliptic equations using nonconforming elements. *Mathematics of Computation*, 31(137):45–59, 1977.

[5] F. Bassi and S. Rebay. A high-order accurate discontinuous finite element method for the numerical solution of the compressible navier–stokes equations. *Journal of computational physics*, 131(2):267–279, 1997.

[6] P. Bastian. Numerical computation of multiphase flow in porous media. 1999.

[7] P. Bastian. Higher order discontinuous galerkin methods for flow and transport in porous media. In *Challenges in Scientific Computing-CISC 2002*, pages 1–22. Springer, 2003.

[8] P. Bastian. A fully-coupled discontinuous galerkin method for two-phase flow in porous media with discontinuous capillary pressure. *Computational Geosciences*, 18(5):779–796, 2014.

[9] S. Brdar, A. Dedner, and R. Klöfkorn. Compact and stable discontinuous galerkin methods for convection-diffusion problems. *SIAM Journal on Scientific Computing*, 34(1):A263–A282, 2012.

[10] R. Brooks and A. Corey. Hydraulic properties of porous media. *Hydrology Papers. Colorado State University*, (3), 1964.

[11] B. Cockburn and C.-W. Shu. The local discontinuous galerkin method for time-dependent convection-diffusion systems. *SIAM Journal on Numerical Analysis*, 35(6):2440–2463, 1998.

[12] A. Dedner, R. Klöfkorn, M. Nolte, and M. Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the dune-fem module. *Computing*, 90(3-4):165–196, 2010.

[13] D. A. Di Pietro and A. Ern. *Mathematical aspects of discontinuous Galerkin methods*, volume 69. Springer Science & Business Media, 2011.

[14] J. Douglas and T. Dupont. Interior penalty procedures for elliptic and parabolic galerkin methods. In *Computing methods in applied sciences*, pages 207–216. Springer, 1976.

[15] Y. Epshteyn and B. Rivière. Fully implicit discontinuous finite element methods for two-phase flow. *Applied Numerical Mathematics*, 57(4):383–401, 2007.

[16] A. Ern, I. Mozolevski, and L. Schuh. Discontinuous galerkin approximation of two-phase flows in heterogeneous porous media with discontinuous capillary pressures. *Computer methods in applied mechanics and engineering*, 199(23):1491–1501, 2010.

[17] J. S. Hesthaven and T. Warburton. *Nodal discontinuous Galerkin methods: algorithms, analysis, and applications*. Springer Science & Business Media, 2007.

[18] W. Klieber and B. Riviere. Adaptive simulations of two-phase flow by discontinuous galerkin methods. *Computer methods in applied mechanics and engineering*, 196(1):404–419, 2006.

[19] W. H. Reed and T. Hill. Triangularmesh methodsfor the neutrontransportequation. *Los Alamos Report LA-UR-73-479*, 1973.

[20] J.-F. Remacle, J. E. Flaherty, and M. S. Shephard. An adaptive discontinuous galerkin technique with an orthogonal basis applied to compressible flow problems. *SIAM review*, 45(1): 53–72, 2003.

[21] B. Riviere. Numerical study of a discontinuous galerkin method for incompressible two-phase flow. 2004.

[22] M. T. Van Genuchten. A closed-form equation for predicting the hydraulic conductivity of unsaturated soils. *Soil science society of America journal*, 44(5):892–898, 1980.

[23] M. F. Wheeler. An elliptic collocation-finite element method with interior penalties. *SIAM Journal on Numerical Analysis*, 15(1):152–161, 1978.