3

Implicit Discontinuous Galerkin Methods for Transport Equations in Porous Media

ØYSTEIN S. KLEMETSDAL AND KNUT-ANDREAS LIE

Abstract

We explain how you can use discontinuous Galerkin methods to formulate implicit higher-order discretizations of transport equations on stratigraphic and polytopal grids and outline how this is implemented in the `dg` module of the MATLAB Reservoir Simulation Toolbox (MRST).

3.1 Introduction

The single-point upstream-mobility weighting (SPU) scheme, introduced in subsection 9.3.2 and section 9.4 of the MATLAB Reservoir Simulation Toolbox (MRST) textbook [9], is robust and widely used but has certain limitations that can show up when simulating complex flow physics. First of all, the reason the method is robust is because it adds numerical diffusion that inhibits the creation of spurious oscillations near discontinuities and sharp gradients in the solution. In some cases, this can result in excessive smearing of linear or weakly linear waves that can be found as trailing waves in compositional and enhanced oil recovery (EOR) simulations (see Chapters 7 and 8). Fluid mobility and rock wettability typically change significantly across such waves, and smearing them out can lead to largely erroneous predictions of the displacement efficiency of a given injection strategy. A second problem with the SPU scheme is that it is only first-order accurate and thus gives relatively low resolution in smooth parts of the solution. Last, but not least, the method introduces preferential flow directions perpendicular to cell interfaces, which may lead to strong grid-orientation effects for flows with adverse mobility ratios [3, 7] (an examples is shown in the MRST textbook [9, subsection 10.4.2]). These effects result from instabilities in the multiphase flow equations and should not be confused with grid-orientation errors caused by

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

inconsistent discretizations of single-phase fluxes, as discussed in chapter 6 of the MRST textbook [9].

The three problems just mentioned can to a varying degree be mitigated by introducing high-resolution spatial discretizations. Higher-order discontinuous Galerkin (dG) methods [5, 16], first introduced by Reed and Hill [15], are one example of mass-conservative methods that are well-suited to capture sharp displacement fronts without introducing spurious oscillations or excessive numerical smearing. In contrast to other higher-order (high-resolution) methods, such as total-variation diminishing (TVD) [4, 19] and weighted essentially nonoscillatory (WENO) schemes [6, 11], that use a wider stencil of cells to achieve higher order, dG stencils are local to a single cell for all orders and thus give linearized systems that preserve the sparsity structure of the SPU scheme in a blockwise sense.

This chapter outlines how to formulate implicit discontinuous Galerkin methods for a wide class of meshes, including stratigraphic grids and general polytopal meshes. We describe rules for numerical integration, discuss various strategies to diminish spurious oscillations near discontinuities (e.g., slope limiters), and explain in detail how you can use the overall methodology to set up simulations within a sequential solution approach. The cases include the classical Buckley–Leverett problem in 1D, an unstructured polytopal mesh generated by the `upr` module, and a 2D Cartesian case with strongly channelized heterogeneity.

3.2 Model Equations

For the time and spatial scales of interest in the simulation of conventional reservoirs, changes in fluid pressures will usually propagate much faster than changes in fluid phases and chemical components. You can also see this in the model equations. The pressure part of the system is mainly governed by second-order differential operators in space and has a certain elliptic character, whereas the transport of fluid phases and chemical components in most cases, except for capillary-dominated flow, is primarily governed by first-order differential operators in space and has a dominant hyperbolic character. Hyperbolic and elliptic operators are discretized somewhat differently, and in sequential solution methods the overall flow equations are decomposed into a pressure and transport part and solved in sequence; see chapters 8 and 10 of the MRST textbook [9] for details. Discontinuous Galerkin methods can be used to discretize both the elliptic and hyperbolic parts of the model equations, but herein we will only use this formulation for the hyperbolic part. To simplify the discussion, we henceforth assume that fluid pressures, and all quantities that depend on these, are known or have been discretized in some appropriate manner.

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

To develop the discontinuous Galerkin schemes, we start with the conservation equation for fluid phase α, discretized in time by the standard backward Euler method and written in residual form:

$$\mathcal{R}_\alpha^{n+1} = \frac{1}{\Delta t^n}(\mathcal{M}_\alpha^{n+1} - \mathcal{M}_\alpha^n) + \nabla \cdot \vec{\mathcal{F}}_\alpha^{n+1} - \mathcal{Q}_\alpha^{n+1} = 0. \tag{3.1}$$

The three possible phases are the aqueous phase (a), the liquid or oleic phase (ℓ), and the vaporous or gaseous phase (v). For immiscible multiphase flow, the mass term $\mathcal{M}$, flux term $\vec{\mathcal{F}}$, and source/sink term $\mathcal{Q}$ read:

$$\mathcal{M}_\alpha(u) = \phi\rho_\alpha S_\alpha, \quad \vec{\mathcal{F}}(u) = \rho_\alpha \vec{v}_\alpha, \quad \mathcal{Q}_\alpha(u) = \rho_\alpha q_\alpha. \tag{3.2}$$

As a shorthand, we use the (somewhat imprecise) notation u to represent the unknown variables, which for an immiscible multiphase transport problem consist of the phase saturations S_α. The porosity ϕ and the density ρ_α of phase α are generally pressure-dependent quantities that we here assume as given. The volumetric source terms, q_α, are assumed to be known temporal functions for injection wells and functions of the unknown phase saturations S_α for production wells. For the volumetric flow rates, $\vec{v}_\alpha$, usually referred to as the macroscopic Darcy velocity, we use a fractional-flow formulation to derive the following expression (see the MRST textbook [9, subsection 8.3.2]):

$$\vec{v}_\alpha = f_\alpha \left[\vec{v} + \sum_{\beta=a,\ell,v} \lambda_\beta(\vec{G}_\alpha - \vec{G}_\beta) \right]. \tag{3.3}$$

The total velocity $\vec{v}$ generally depends on fluid pressure and saturations but is temporarily assumed to be a known quantity in the next section. The fractional flow f_α and the gravity/capillary flux terms $\vec{G}_\alpha$ are known functions of saturation:

$$f_\alpha = \frac{\lambda_\alpha}{\lambda_a + \lambda_\ell + \lambda_v}, \qquad \vec{G}_\alpha = \rho_\alpha g \mathbf{K}\nabla z + \mathbf{K}\nabla p_c^\alpha. \tag{3.4}$$

To complete the specification: $\mathbf{K}$ is the absolute permeability, g is the gravity acceleration, and λ_α is the phase mobility, defined as the ratio between the relative permeability and the phase viscosity.

Altogether, this gives us a system of m transport equations for m fluid phases, which can be reduced to a system of $m - 1$ equations by using the assumption that the fluids fill the pore volume; i.e., $\sum_\alpha S_\alpha = 1$.

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

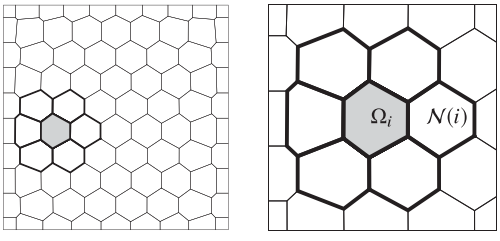
You may wonder why we only use a first-order *temporal* discretization in (3.1). Higher-order temporal discretizations involve explicit terms that limit the time step, and to be able to take very large time steps we thus accept the reduced accuracy of the unconditionally stable backward Euler method. Moreover, the formal spatial order reduces to one half in the presence of discontinuities, which are often the most important part of the solution.

3.3 Discontinuous Galerkin Methods

To also discretize the residual equations in space, we introduce a mesh that partitions the computational domain Ω into nonoverlapping cells $\{\Omega_i\}_{i=1}^{n_c}$. Figure 3.1 shows an example of such a mesh in 2D. The mesh is assumed to have matching faces, so that each cell Ω_i shares a unique face Γ_{ij} with each of its topological neighbors Ω_j; indices of these cells are given by the set $\mathcal{N}(i)$. We use double subscript to denote quantities evaluated at the interface between two cells. We also assume that each cell has a set of associated geometric properties, including cell and face centroids, face areas and normals, and vertices. In MRST, this information is part of the standard grid structure developed to represent general unstructured meshes.

3.3.1 Weak Residual Form

As in any weighted residual method, like a continuous Galerkin method, we start by multiplying the residual equation (3.1) by a test function ψ from a function space V of sufficiently smooth functions and then integrate over each cell Ω_i to obtain




```
G = pebiGrid2D(1/8, [1 ,1]);  % PEBI grid
```

Figure 3.1 Polytopal mesh constructed with the `upr` module. The right figure shows a cell Ω_i in gray and its topological neighbors $\mathcal{N}(i)$ outlined with thick lines. Source code for these plots and the examples in this section can be found in `dgExampleDiscretization.m`.

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

$$0 = \int_{\Omega_i} \mathcal{R}_\alpha \psi \, \mathrm{d}V = \frac{1}{\Delta t} \int_{\Omega_i} (\mathcal{M}_\alpha - \mathcal{M}_\alpha^n)\psi \, \mathrm{d}V + \int_{\Omega_i} (\nabla \cdot \vec{\mathcal{F}}_\alpha)\psi \, \mathrm{d}V - \int_{\Omega_i} \mathcal{Q}_\alpha \psi \, \mathrm{d}V. \tag{3.5}$$

Here, we have dropped superscript $n + 1$ for the next time step. To move the derivative in the flux term to the test function, we write the weighted divergence of the flux as a sum of a surface integral and a volume integral

$$\int_{\Omega_i} (\nabla \cdot \vec{\mathcal{F}}_\alpha)\psi \, \mathrm{d}V = \int_{\partial\Omega_i} (\vec{\mathcal{F}}_\alpha \psi) \cdot \vec{n} \, \mathrm{d}S - \int_{\Omega_i} \vec{\mathcal{F}}_\alpha \cdot \nabla\psi \, \mathrm{d}V. \tag{3.6}$$

In finite element theory, it is common to replace integrals involving test functions by the equivalent L^2 inner products

$$\big(h, \psi\big)_{\Omega_i} = \int_{\Omega_i} h\psi \, \mathrm{d}V, \qquad \big(h, \psi\big)_{\partial\Omega_i} = \int_{\partial\Omega_i} h\psi \, \mathrm{d}S. \tag{3.7}$$

In this notation, the *weak form* of the residual equation (3.1) reads

$$\begin{aligned} \big(\mathcal{R}_\alpha(u), \psi\big)_{\Omega_i} = \frac{1}{\Delta t}\big(\mathcal{M}_\alpha(u) - \mathcal{M}_\alpha(u^n), \psi\big)_{\Omega_i} \\ + \big(\vec{\mathcal{F}}_\alpha(u) \cdot \vec{n}, \psi\big)_{\partial\Omega_i} - \big(\vec{\mathcal{F}}_\alpha(u), \nabla\psi\big)_{\Omega_i} - \big(\mathcal{Q}_\alpha(u), \psi\big)_{\Omega_i} = 0. \end{aligned} \tag{3.8}$$

3.3.2 Basis Functions

We obtain a fully discrete weak formulation of (3.1) by replacing the (infinite-dimensional) function space V by a finite-dimensional subspace V_h of functions that are piecewise polynomials on each cell but discontinuous across cell interfaces. We may represent this space locally on a cell Ω_i by a set of basis functions $\{\psi_i^k\}_{k=1}^{n_{\mathrm{dof}}}$, which we also use to express an unknown function $u \in V_h$ in Ω_i,

$$u = u(\vec{x}) = \sum_{k=1}^{n_{\mathrm{dof}}} u_i^k \psi_i^k(\vec{x}). \tag{3.9}$$

We refer to u_i^k as the kth degree of freedom of u in Ω_i. There are several different ways to choose basis functions. For linear problems, one can use *orthonormal* basis functions, for which $(\psi_i^k, \psi_i^\ell)_{\Omega_i} = \delta_{k\ell}$, to ensure that the mass matrix is diagonal. Our flow equations are generally nonlinear, so using an orthonormal basis will not reduce sparsity. To get a dG method with polynomial basis functions of order k or lower, we instead define the basis to be the set of all functions of the form

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

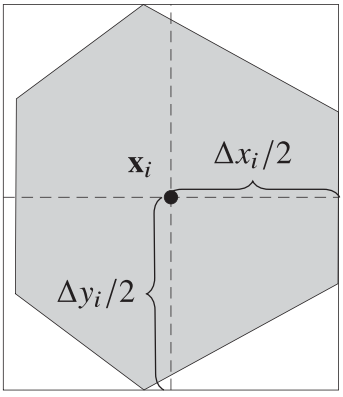


Figure 3.2 Polygonal cell Ω_i from Figure 3.1. The bounding box has dimensions $\Delta\vec{x}_i = (\Delta x_i, \Delta y_i)$, is aligned with the axial directions, and is computed so that the centroid of the box coincides with the centroid $\vec{x}_i$. Basis functions are constructed to be symmetric about the dashed coordinate axes through $\vec{x}_i$. (Hence the small gap at the left edge.)

$$\psi_i^j(\vec{x}) = \begin{cases} \ell_p\left(\frac{x-x_i}{\Delta x_i/2}\right)\ell_q\left(\frac{y-y_i}{\Delta y_i/2}\right)\ell_r\left(\frac{z-z_i}{\Delta z_i/2}\right) & \text{if } \vec{x} \in \Omega_i, \\ 0 & \text{otherwise,} \end{cases} \tag{3.10}$$

where $p, q, r \geq 0$ and $p + q + r \leq k$. Here, ℓ_p is the pth Legendre basis function, $\vec{x}_i = (x_i, y_i, z_i)$ is the cell centroid, and $\Delta\vec{x}_i = (\Delta x_i, \Delta y_i, \Delta z_i)$ are the dimensions of the smallest bounding box aligned with the coordinate axes that contain Ω_i so that its centroid coincides with $\vec{x}_i$; see Figure 3.2. The dimensions of each bounding box are computed using `computeCellDimensions(G)`.

Using kth-order polynomial basis functions gives a dG method of formal order $k + 1$, which we denote by dG(k). The total number of basis functions for a dG(k) method in d spatial dimensions equals

$$n_{\text{dof}} = \binom{k+d}{d} = \frac{(k+d)!}{d!\,k!}. \tag{3.11}$$

We now have a fully discrete formulation for the weak bilinear form of (3.1):

For all $i = 1 \ldots n_c$, find $(u_i^1, \ldots u_i^{n_{\text{dof}}}) \in \mathbb{R}^{n_{\text{dof}}}$ so that

$$\left(\mathcal{R}_\alpha\left(\sum_{\ell=1}^{n_{\text{dof}}} u_i^\ell \psi_i^\ell\right), \psi_i^k\right)_{\Omega_i} = 0 \quad \forall k = 1, \ldots, n_{\text{dof}}. \tag{3.12}$$

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

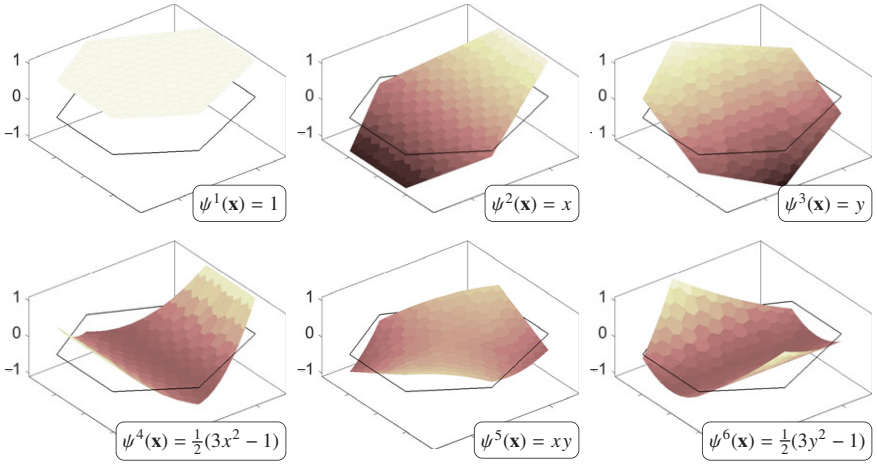


Figure 3.3 The six Lengendre-type basis functions in a dG(2) method for the cell indicated in Figure 3.1. Note that the basis functions are expressed in reference coordinates. (Color shows function values.)

Discontinuous Galerkin discretizations are implemented in MRST in the class `DGDiscretization`, which can be found in the `dg` module:

```
mrstModule add dg                          % Load module
G    = computeCellDimensions(G);           % Compute cell dimensions
disc = DGDiscretization(G, 'degree', 2); % Construct dG(2) discretization
```

By default, this class constructs basis functions of the form (3.10):

```
>> disp(disc.basis)
        psi: {6×11 cell}
    gradPsi: {6×11 cell}
     degree: 2
          k: [6×12 double]
       nDof: 6
       type: 'legendre'
```

```
>> disp(disc.basis.psi{3})
    Polynomial with properties:

      k: [0 1]
      w: 1
    dim: 2
```

The `basis` structure holds basis functions `psi` and their gradients `gradPsi`, in addition to other associated quantities. The basis functions are conveniently implemented in the `Polynomial` class, which defines the associated exponents `k` and coefficients `w`. This class has overloaded operators for elementary algebraic operations, as well as derivatives, gradients, and tensor products, which enables simple generation of basis functions of arbitrary order. Figure 3.3 depicts six basis functions for dG(2), plotted using `plotDGBasis(G,disc.basis,c)` for a given cell `c`.

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

3.3.3 Numerical Integration

Closed-form expressions for the bilinear forms in (3.8) are not readily available for unstructured grids and are not convenient to use when integrated relative permeabilities, pressure–volume–temperature functions, and other physical properties are represented in tabulated form. We must therefore resort to numerical integration using cubature rules [1]. We make the simplifying assumptions that all grids are polytopal; i.e., they have planar faces. The resulting rules can also be applied as an approximation for stratigraphic grids but need to be replaced by more accurate rules for bilinear faces that are strongly curved.

Given a cubature rule with a set of points $\vec{x}_1, \ldots, \vec{x}_{n_p}$ and corresponding weights $w_1, \ldots w_{n_p}$, the bilinear forms over Ω_i are approximated as

$$(\mathcal{M}_\alpha(u), \psi)_{\Omega_i} \approx |\Omega_i| \sum_k w_k \mathcal{M}_\alpha(u(\vec{x}_k))\psi(\vec{x}_k). \tag{3.13}$$

A cubature rule is of *precision* k if it is exact for all polynomials of order k or less. This means that the weights w_k must sum to 1 for each cell – a convenient check for any cubature rule. For a dG(k) method, we have to integrate polynomials of total order $2k$, and we must thus ensure that the cubature rule has precision $2k$ or larger.

Perhaps the simplest way to construct cubature rules for a general polyhedral cell is to subdivide into simplices, for which well-known cubature rules exist. This approach is implemented in the `TriangleCubature` (2D) and `TetrahedronCubature` (3D) classes, which both inherit from the `Cubature` class. The underlying cubature rules in these classes are taken from [18] and are the default used to evaluate the bilinear forms over cells in the dG implementation. We can visualize the cubature using `plotCubature(G,i)`; the left plot in Figure 3.4 shows the result.

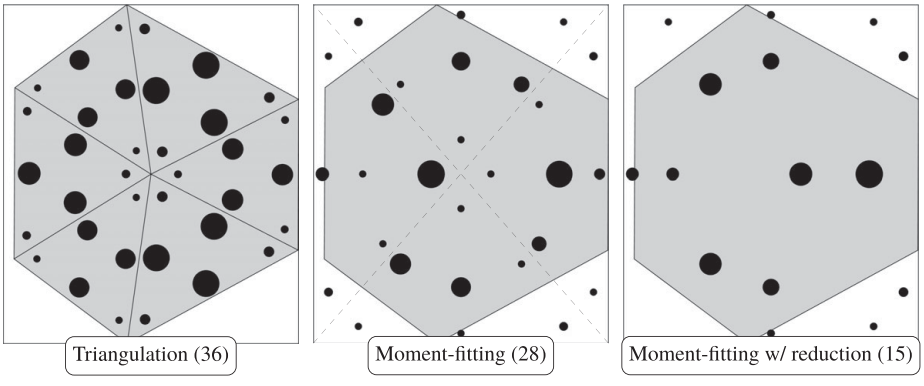
To evaluate an integral numerically, we get the cubature points and weights, transform the coordinates from physical to reference space, evaluate the integrand in the points, and take the weighted sum. As an example, the integral of the basis functions over Ω_i is

```
[W, x, w, cells] = disc.cellCubature.getCubature(i, 'cell');
x = disc.cubature.transformCoords(x, cells);
cellfun(@(psi) W*psi(x), disc.basis.psi) % approximate integral computed by
                                         % matrix-vector multiplication
```

```
ans =
   1.0000   -0.0000    0.0000   -0.1178   -0.0006   -0.1948
```

Refering back to Figure 3.3, we see that the first basis function is constant and equal to 1. This means that the first integral is simply the sum of the quadrature points, which should be equal to 1. The second and third basis functions are linear, and their integrals over Ω_i vanish by construction.

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press




```
triCubature    = disc.cellCubature;
mfCubature_nr = MomentFitting2DCubature(G, 2*degree, 'chunkSize', 1, 'reduce', false);
mfCubature     = MomentFitting2DCubature(G, 2*degree, 'chunkSize', 1);
subplot(1,3,1), plotCubature(G, triCubature,    c, 'faceColor', gray)
subplot(1,3,2), plotCubature(G, mfCubature_nr, c, 'faceColor', gray)
subplot(1,3,3), plotCubature(G, mfCubature,     c, 'faceColor', gray)
```

Figure 3.4 Three different cubatures for numerical evaluation of integrals over Ω_i: `TriangleCubature`, based on triangulation. The cubature point markers are scaled by the corresponding integration weights. (Source code: `dgExampleDiscretization.m`.)

Triangulation is clearly inefficient, because it results in a cubature rule with significantly more points than strictly needed for the cubature rule to be correct. This is particularly true for implicit simulations with automatic differentiation, in which evaluation in a cubature point consists of potentially costly operations involving the Jacobians. Efficient cubatures can be constructed using moment fitting [12], which is implemented in the `MomentFitting2DCubature` and `MomentFitting3DCubature` classes, which rely on functionality from the `vemmech` module. In this approach, we use a set of m predefined points and define the cubature rule for Ω_i as the solution to the following linear system (omitting cell subscript i):

$$\begin{bmatrix} \psi^1(\vec{x}_1) & \cdots & \psi^1(\vec{x}_m) \\ \vdots & \cdots & \vdots \\ \psi^n(\vec{x}_1) & \cdots & \psi^n(\vec{x}_m) \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} = \frac{1}{|\Omega|} \begin{bmatrix} \int_\Omega \psi^1 \, \mathrm{d}V \\ \vdots \\ \int_\Omega \psi^n \, \mathrm{d}V \end{bmatrix}, \quad \text{or} \quad \Psi \vec{w} = \vec{b}.$$

Here, $\{\psi^i\}_{i=1}^n$ is the polynomial basis (e.g., Legendre) for the space of polynomials the cubature is intended for, with $n < m$. The right-hand side can be computed by any suitable means – typically a triangulation quadrature. The corresponding weights are then found as the constrained linear least-squares solution to the underdetermined minimization problem

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

$$\min(\Psi\vec{w} - \vec{b}) \quad \text{such that} \quad \vec{w} \geq 0. \tag{3.14}$$

The quadrature is reduced by iteratively removing the least significant point having the smallest value of $\sum_k \psi^k(\vec{x}_i)$ and recomputing the solution to (3.14). If the residual $\|\Psi\vec{w} - \vec{b}\|$ is sufficiently small (10^{-14} in our implementation), the point is removed. We repeat the process until no points can be removed. The middle and right plots in Figure 3.4 illustrate a `MomentFitting2DCubature` without and with reduction. Even without reduction, we see that the cubature uses significantly fewer points (28) than `TriangleCubature` (36). With reduction, the cubature uses only 15 points, which equals the number of basis functions needed to represent all polynomials of order less than or equal to 2×2. Cubatures are computed once in a preprocessing step.

3.3.4 Evaluating the Interface Flux

The integrand $\vec{\mathcal{F}}_\alpha \cdot \vec{n}$ in the second term of (3.8) is in principle discontinuous in all points at $\partial\Omega_i$ and consequently requires a more delicate treatment. First, we write the bilinear form as a sum of integrals over all interfaces $\{\Gamma_{ij}\}_{j\in\mathcal{N}(i)}$ of Ω_i,

$$(\vec{\mathcal{F}}_\alpha(u) \cdot \vec{n}, \psi)_{\partial\Omega_i} = \sum_{j\in\mathcal{N}(i)} (\vec{\mathcal{F}}_\alpha(u) \cdot \vec{n}, \psi)_{\Gamma_{ij}}. \tag{3.15}$$

Looking back at the phase flux in (3.3) and (3.4), we see that it consists of a combination of saturation-dependent terms and terms that are either constant or depend on pressure. The total Darcy flux $\vec{v} \cdot \vec{n}$ and the gravity/capillary terms $\vec{G}_\beta \cdot \vec{n}$ across each interface are discretized using a standard two-point scheme, giving constant values for each interface Γ_{ij}; see [9, section 9.4] for details. For the saturation-dependent mobilities, we use single-point upstream weighting in each cubature point:

$$\{\lambda_\alpha(\vec{x}); \vec{v}_\alpha \cdot \vec{n}\}_{ij} = \begin{cases} \lambda_i(\vec{x}) & \text{if } \vec{v}_\alpha(\vec{x}) \cdot \vec{n} \geq 0, \\ \lambda_j(\vec{x}) & \text{otherwise.} \end{cases} \tag{3.16}$$

Note that this involves an *implicit* upstream definition, because the direction of $\vec{v}_\alpha$ depends on the unknowns u. Fortunately, it is possible to replace the implicit definition by an explicit evaluation procedure outlined in [9, section 9.4]. Applying this at each cubature point, we can approximate the interface bilinear forms as

$$(\vec{\mathcal{F}}_\alpha(u) \cdot \vec{n}, \psi)_{\Gamma_{ij}} \approx |\Gamma_{ij}| \sum_k w_k \{\vec{\mathcal{F}}_\alpha(u(\vec{x}_k)), v_\alpha\}_{ij} \psi(\vec{x}_k). \tag{3.17}$$

Figure 3.5 illustrates the cubature rule between Ω_i and one of its neighbors.

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

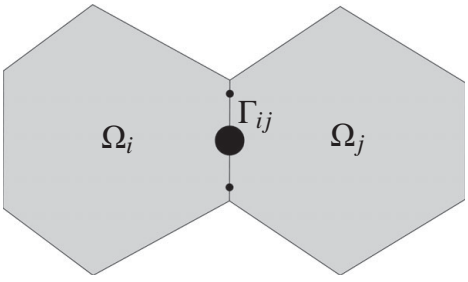


Figure 3.5 Cubature over the interface Γ_{ij} between cells Ω_i and Ω_j. The integrand is evaluated from Ω_i or from Ω_j depending on the direction of the flux in the quadrature the point. As in Figure 3.4, the size of the points are proportional to the corresponding integration weights.

3.3.5 Velocity Interpolation

The only remaining term that we have not yet discussed is the volume integral of $\vec{\mathcal{F}}_\alpha \cdot \nabla\psi$, which is generally nonzero for linear or higher-order basis functions. The two-point scheme we just used to discretize the total Darcy flux and the gravity/capillary flux only gives us values v_{ij} and G_{ij} associated with each interface. The constant gravity contribution $g\mathbf{K}\nabla z$ is evaluated exactly. For the total Darcy velocity, we apply a simple scheme, inspired by the mimetic finite-difference method discussed in section 6.4 of the MRST textbook [9], to define a constant velocity $\vec{v}$ inside cell Ω_i that is consistent with the volumetric interface fluxes v_{ij}:

$$\vec{v}_i = \frac{1}{|\Omega_i|} \sum_{j \in \mathcal{N}(i)} v_{ij}(\vec{x}_{ij} - \vec{x}_i). \tag{3.18}$$

Here, $\vec{x}_i$ and $\vec{x}_{ij}$ refer to the centroids of Ω_i and Γ_{ij}, respectively. The same approach is applied to the capillary term. (Consult [8] for more accurate interpolation methods.)

3.3.6 Limiters

As any other high-order method, dG(k) tends to exhibit spurious oscillations near discontinuities for $k \geq 1$. To get a *high-resolution method* that has at least second-order spatial accuracy in smooth parts of the solution and is free of oscillations near discontinuities, we need to add a nonlinear mechanism, called a *limiter*, that effectively introduces enough numerical diffusion to dampen oscillations. We use two such limiters by default: a total variation bounded (TVB) limiter [17] that adjusts the gradient whenever the jump across an interface is greater than zero and a physical limiter that scales the solution to be within physical limits inside

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

each cell. In our experience, these limiters must be applied *after* the time step, because applying limiters in between nonlinear iterations tends to severely impede convergence.

Finite-volume (FV) limiter: After a nonlinear iteration, MRST does not necessarily update the unknowns directly but performs a number of operations on the increments calculated by the Newton solver before they are applied to update the unknowns. This involves restricting the cell averages to stay within physical limits, in addition to damping strategies such as chopping with absolute or geometric penalties; see [9, sections 12.2 and 12.3]. For miscible flow, we also handle complex physical processes such as phase appearance/disappearance. In the dG implementation, we use some of these ideas: First, absolute penalties are used when updating the unknowns. If the underlying base model uses a maximum absolute update $\Delta u_{\max}$ for the unknown u, we apply a maximum update of $\Delta u_{\max}/n_{\text{dof}}$ for the dG unknowns. Then, after updating the dG unknowns u_i^k of u_i, we apply the following limiter:

$$\mathcal{L}_{\text{FV}}(u_i) = \sum_{k=1}^{n_{\text{dof}}} \left(u_i^k + \left(\frac{\delta \bar{u}_i}{\Delta \bar{u}_i} - 1 \right) \Delta u_i^k \right) \psi_i^k. \tag{3.19}$$

Here, $\Delta \bar{u}_i$ is the update of the cell mean $\bar{u}_i$ resulting from the updated dG unknowns, and $\delta \bar{u}_i$ is the actual update in $\bar{u}_i$ after passing $\Delta \bar{u}_i$ to the update function of the base model. In effect, this limiter scales the updates to the higher-order degrees of freedom so that the mean value is updated according to the same rules as the base model, honoring physical limits.

TVB limiter: The TVB limiter [17] is applied after nonlinear convergence and adjusts the gradient in all cells where the jump across the interface to any neighboring cell exceeds a given tolerance ϵ_{TV}. To explain the limiter, let us renumber our basis functions so that ψ^0 is the constant function, $\psi^{1,1}, \ldots, \psi^{1,d}$ are the linear test functions, etc. With this, the limiter reads

$$\mathcal{L}_{\text{TV}}(u_i) = \begin{cases} u_i^0 \psi_i^0 + \hat{u}_i^1 \psi_i^{1,1} + \cdots + \hat{u}_i^d \psi_i^{1,d}, & \text{if } |[u]| > \epsilon_{\text{TV}} \text{ on } \partial\Omega_i \\ u_i, & \text{otherwise,} \end{cases} \tag{3.20}$$

$$\hat{u}_i^k = \text{minmod}\big(u_i^k, \{\nabla u_{ipq}^k : p \neq q,\ p, q \in \mathcal{N}(i)\}\big).$$

The minmod operator is a standard limiter, defined as

$$\text{minmod}\big(v_1, \ldots v_n\big) = \begin{cases} \min(v_1, \ldots, v_n) & \text{if all } v_i > 0, \\ \max(v_1, \ldots, v_n) & \text{if all } v_i < 0, \\ 0 & \text{otherwise.} \end{cases} \tag{3.21}$$

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

In other words, if the jump across $\partial\Omega_i$ is larger than the tolerance ϵ_{TV}, the limiter adjusts the degrees of freedom associated with linear basis functions (e.g., only ψ^2 and ψ^3 in Figure 3.3), whereas all higher-order degrees of freedom are set to zero. If the jump is less than the tolerance, the solution in Ω_i remains unchanged. The gradient is adjusted by comparing the original degree of freedom u_i^k in the cell with approximations of the derivative obtained from the mean value of Ω_i and the mean values in neighboring cells Ω_j, $j \in \mathcal{N}(i)$; see [14, 20]. This reconstruction is relatively simple on rectilinear grids but complicated on polytopal grids. In our implementation, we therefore borrow a technique from the particular WENO discretization in MRST that obtains a number of gradient approximations $\nabla u_{ipq} = (u_{ipq}^1, \ldots, u_{ipq}^d)$ by building planar reconstructions using triplets of values (u_i^0, u_p^0, u_q^0) from Ω_i and two of its neighbors, Ω_p and Ω_q, for $p, q \in \mathcal{N}(i)$; see [10] for more details. The kth degree of freedom $\hat{u}_i^k$ is then taken as the minmod of the original value u_i^k and all of the derivative approximations. Note that this limiter adjusts the gradient in each coordinate direction independently.

Physical limiter: To ensure that each unknown u value stays within its physical bounds, $[u_{\min}, u_{\max}]$, we also use a physical limiter [2] that scales the solution after nonlinear convergence,

$$\begin{aligned} \mathcal{L}_{\text{ph}}(u_i) &= u_i^0\psi_i^0 + \theta_i\big(u_i - u_i^0\psi_i^0\big), \\ \theta_i &= \min\left\{\left|\frac{u_i^0 - u_{\min}}{u_i^0 - \min(u_i)}\right|, \left|\frac{u_{\max} - u_i^0}{\max(u_i) - u_i^0}\right|, 1\right\}. \end{aligned} \tag{3.22}$$

Here, $\min(u_i)$ and $\max(u_i)$ are measured as the minimum and maximum value of u at all cubature points inside or on the faces of Ω_i. This limiter ensures that the minimum and maximum values of u are within the physical bounds at the end of each timestep. During the nonlinear solution process, however, we allow the solution to violate the physical bounds, because this generally tends to aid nonlinear convergence.

Observant readers may have noticed that because our basis functions are not orthogonal by definition for general cell geometries, applying a limiter to adjust the higher-order degrees of freedom may have the unfortunate side effect of also changing the cell mean $\bar{u}_i$. However, because all linear basis functions are orthogonal by construction, we can avoid this by replacing u_i^0 with the cell mean $\bar{u}_i$ in (3.20) and (3.22). Likewise, applying the TVB and physical limiters after nonlinear convergence effectively changes the mass flux out of a cell and may therefore introduce a mass error. In our experience, however, this is necessary for a robust, implicit dG implementation, and the mass error does not seem to grow unacceptably large.

This concludes the description of the `dg` module. Figure 3.6 illustrates the solution of one timestep with the module.

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

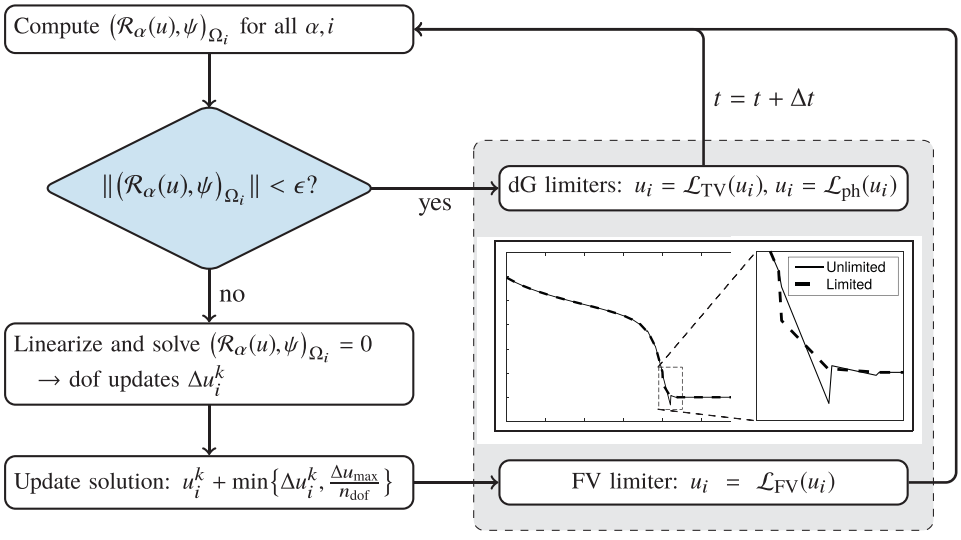


Figure 3.6 Flowchart describing the steps necessary to advance the solution one timestep. To compute the weak residual, we evaluate the conservation equations and basis functions in all quadrature points of all cells. If the residual exceeds the nonlinear tolerance ϵ, we linearize the system of equations using automatic differentiation (see Chapter 6) and solve to find the dG updates. The solution is then updated using the FV limiter (3.19). After convergence, we apply the TVB limiter (3.20) and the physical limiter (3.22) to dampen spurious oscillations and ensure that the solution is within physical limits.

3.4 Numerical Examples

In the following, we will go through a few numerical examples to demonstrate how you can use the dG method just outlined, as implemented in the `dg` module of MRST, to solve transport problems.

3.4.1 1D Buckley–Leverett Displacement

The first example considers the canonical Buckley–Leverett displacement in a 1D domain with unit permeability and porosity (see `dgExampleBLDisplacement.m` for full code). We start by constructing a standard MRST grid structure and extending it with additional geometric information required by the dG discretization:

```
G    = computeGeometry(cartGrid([30,1]));
G    = computeCellDimensions(G);
rock = makeRock(G, 1, 1);
```

That is, we extend the grid to include a mapping from cells to their bounding vertices, represented as two fields `G.cells.nodes` and `G.cells.nodePos` that

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

are analogous to the fields `faces` and `facePos` that map from cell to bounding faces. In addition, each cell has an associated bounding box represented by the fields `xMin` and `xMax` plus the redundant dimension, `dx`. Similar bounding boxes are added to each face as part of the `G.faces` structure. In addition, `G.faces` contains two function handles, `phys2ref` and `ref2phys`, that map coordinates from physical space to reference space and back again. These are trivial for a 1D mesh but inevitable for general unstructured meshes.

We assume a two-phase model with water and oil, in which both fluid phases have Corey relative permeability curves with quadratic exponent and unit viscosity:

```
fluid = initSimpleADIFluid('phases', 'WO' , 'n', [2,2], ...
                           'mu', [1,1],  'rho', [1,1]);
model  = GenericBlackOilModel(G, rock, fluid, 'gas', false);
```

The reservoir is initially filled with oil, and when water is injected from the left we get the classical Buckley–Leveret profile consisting of a leading shock, followed by a rarefaction wave. To simulate the problem using the standard finite-volume discretization in MRST, we first set the flux explicitly in the initial state:

```
state0 = initResSol(G, 1, [0,1]);
state0.flux(1:G.cells.num+1) = 1;
```

and then define a `TransportModel`, which is implemented as a wrapper around the base model (in this case, the `GenericBlackOilModel`). This ensures that we handle the physics in the exact same way as in fully implicit simulations:

```
tmodel           = TransportModel(model);
[~, stFV, repFV] = simulateScheduleAD(state0, tmodel, schedule);
```

To simulate the transport problem with dG, we use the class `TransportModelDG`, which is a subclass of `TransportModel` from `blackoil-sequential`. The class constructor takes optional arguments for `DGDiscretization`, such as degree:

```
tmodelDG0 = TransportModelDG(model, 'degree', 0);
[~, stDG0, repDG0] = simulateScheduleAD(state0, tmodelDG0, schedule);
```

The transport model with dG holds the discretization, along with the base model (or `parentModel`), etc.

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

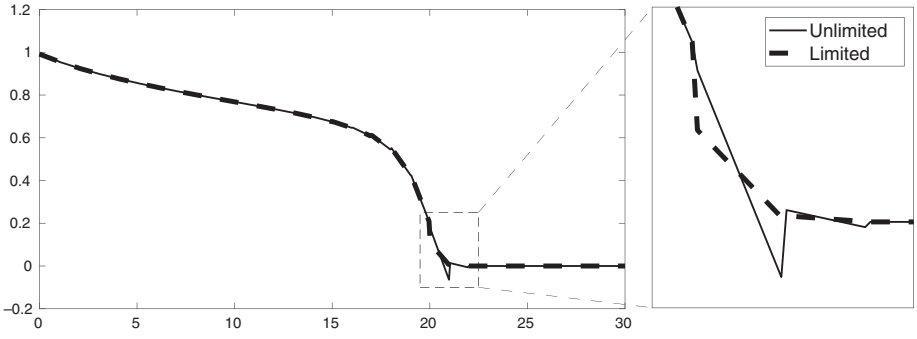


Figure 3.7 Limited and unlimited solutions for the Buckley–Leverett example.

```
>> disp(tmodelDG0)
  TransportModelDG with properties:
          discretization: [1×1 DGDiscretization]
                limiters: [2×1 struct]
          storeUnlimited: 0
             formulation: 'totalSaturation'
             parentModel: [1×1 GenericBlackOilModel]
                  ...
```

Setting up the second-order dG(1) scheme is completely analogous, except we now need to specify the orders of the basis functions in each coordinate direction. Because we use a 2D grid with a single cell in the y-direction to simulate the 1D case, the test functions are set to be constant in this direction:

```
tmodelDG1 = TransportModelDG(model, 'degree', [1,0]);
tmodelDG1.storeUnlimited = true; % Store the unlimited state in each step
fn = plotLimiter(tmodelDG1, 'plot1d', true, 'n', 500); % afterStepFn
  [~, stDG1, repDG1] = ...
  simulateScheduleAD(state0, tmodelDG1, schedule, 'afterStepFn', fn);
```

To see the effect of the limiters, we specify that the solver should store the unlimited solution and then use the optional argument `afterStepFn` in `simulateScheduleAD` to plot the unlimited and limited solution after each timestep. Figure 3.7 shows how the limiter adjusts the slope of the dG(1) solution.

In principle, the `dg` module supports arbitrary order, as long as we can also provide a sufficiently accurate cubature rule. Figure 3.8 shows the solutions obtained using dG(0), dG(1), dG(2), and the standard SPU scheme, compared to the exact solution. Notice that using constant basis functions in dG(0) gives a stair-stepped solution and that this coincides exactly with the SPU solution. Whereas using dG(2)

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

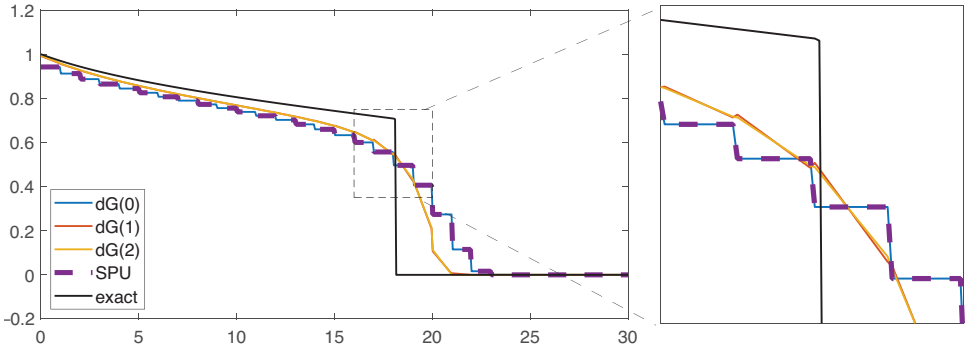


Figure 3.8 Solutions computed using dG(0), dG(1), dG(2), and FV together with the exact solution for the Buckley–Leverett example.

gives a slightly smoother solution than dG(1), we see that most of the accuracy gained in this example comes from introducing linear basis functions in dG(1).

Readers familiar with *explicit* dG methods may be somewhat disappointed by the resolution of the higher-order dG schemes in this example. The reason is that we use a *first-order implicit* temporal discretization to escape the severe timestep restriction imposed on explicit schemes. This introduces significant numerical smearing that counteracts the effect of the high-resolution spatial discretization. Chapter 5 shows an example of how an *adaptive-implicit method* (AIM) solution method can be combined with a high-resolution spatial discretization to significantly reduce the numerical smearing. Another method to reduce smearing is to use a more compressive limiter – i.e., a limiter that tends to choose steeper reconstructions – but this runs the risk of turning smooth bumps into discontinuities.

3.4.2 Smearing of Trailing Waves

In the Introduction we mentioned more accurate resolution of trailing waves as a motivation for introducing a higher-order spatial discretization. Such waves typically arise in multicomponent systems (see Chapter 7 for examples from surfactant–polymer flooding). We can mimic the same behavior by considering a Buckley–Leverett problem with an piecewise linear approximation to the fractional-flow function. Here, we use an approximation consisting of five line segments.

As explained in section 8.4 of the MRST textbook [9], the solution of the Riemann problem underlying the Buckley–Leverett displacement theory is solved by determining the concave envelope of the fractional-flow function between the right state (the resident fluid) and the left state (the injected fluid). Figure 3.9 shows that this envelope consists of three linear segments: one that lies above the original

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

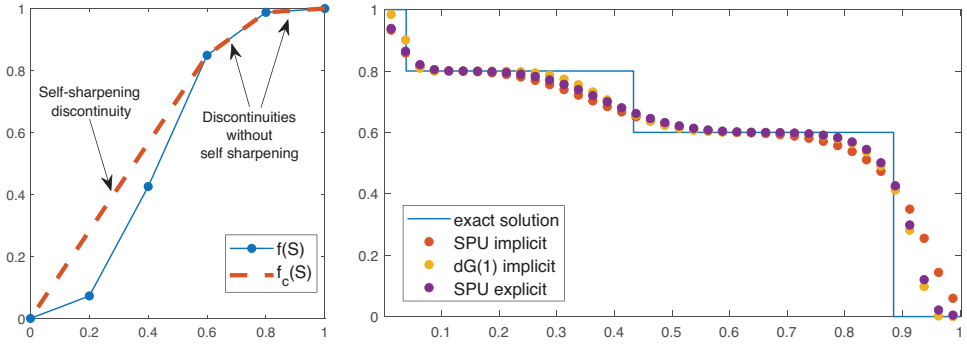


Figure 3.9 Resolution of trailing waves without self sharpening. The left plot shows the piecewise linear flux function f and its concave envelope f_c that defines the Riemann solution. (Source code: `dgExampleTrailing.m`.)

fractional-flow function f and two that coincide with segments of f. Effectively, this means that the characteristic profile, consisting of a leading shock and a trailing rarefaction wave, turns into a shock followed by two trailing (linear) discontinuities that contain no self-sharpening.

The right plot in Figure 3.9 shows solutions computed with three different schemes. As expected, dG(1) with implicit discretization delivers approximately the same resolution of the leading discontinuity as the explicit SPU scheme but better resolution of both trailing waves. To explain this, we can look at the analysis of truncation errors for the SPU scheme for linear waves; e.g., as outlined in subsection 9.3.2 of the MRST textbook [9]. The numerical smearing of the explicit SPU scheme *increases* with decreasing Courant number but *decreases* with decreasing Courant number for the implicit variant, thus giving equal resolution of the slowest wave.

3.4.3 Inverted Five-Spot Pattern on a Perpendicular Bisector Grid

The next example is a 2D scaled-down version of the SPE10 benchmark case. Using a function from the `spe10` module, we construct a fully implicit simulation model for an inverted five-spot posed on a subset of the 13th layer from the Tarbert formation:

```
[init0, model0, schedule0] = ...
   setupSPE10_AD('layers', 13, 'J', (1:110) + 30);
```

However, instead of simulating directly on this 60×110 Cartesian model, we use the `pebiGrid2D` function from the `upr` module (see Chapter 1) to construct an unstructured mesh with approximately 20 cells in each direction. In addition, we

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

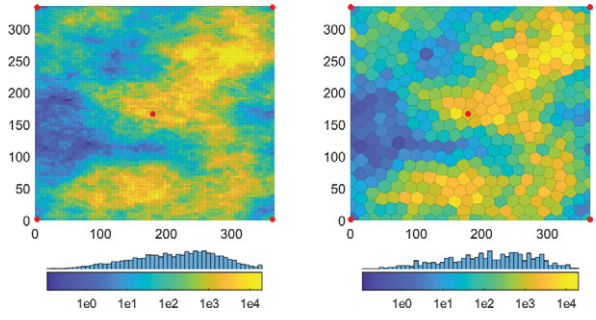


Figure 3.10 Setup of a coarse perpendicular bisector grid based on a 60 × 110 subsample from the 13th layer of the SPE10 benchmark. Well positions are marked in red.

impose a certain degree of near-well refinement so that cells containing wells are 40% the size of the cells in the middle of the reservoir. This gives a polytopal mesh with 449 cells; see Figure 3.10. You can find the necessary details of how to construct this coarse mesh and set up the corresponding wells, petrophysical and initial data, and modified simulation schedule in the script `dgExampleIFS.m`. (Notice that because petrophysical properties are sampled and not *upscaled*, it will not make sense to compare results directly with the fine-scale model.) To simulate the model on the coarse grid, we start by defining a two-phase, fully implicit model:

```
model  = GenericBlackOilModel(G, rock, model0.fluid, 'gas', false);
```

We then use functionality from the `blackoil-sequential` module to set up a sequential simulation model consisting of a standard pressure solver and an implicit dG(0) transport solver:

```
pmodel = PressureModel(model);                    % pressure model
tmodel = TransportModelDG(model, 'degree', 0);  % transport model
seqmodel = SequentialPressureTransportModel(pmodel, tmodel,'parentModel', model);
```

which we can simulate using the standard simulator interface from `ad-core`:

```
[ws, states, reports] = simulateScheduleAD(state0, seqmodel, schedule);
```

The setup for dG(1) is completely analogous. Figure 3.11 shows 3D plots of the water saturation at the end of each simulation. Unlike a standard finite-volume method, which only computes averaged values per cell, the dG methods give point-wise solutions that are continuous inside each cell but discontinuous across cell

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

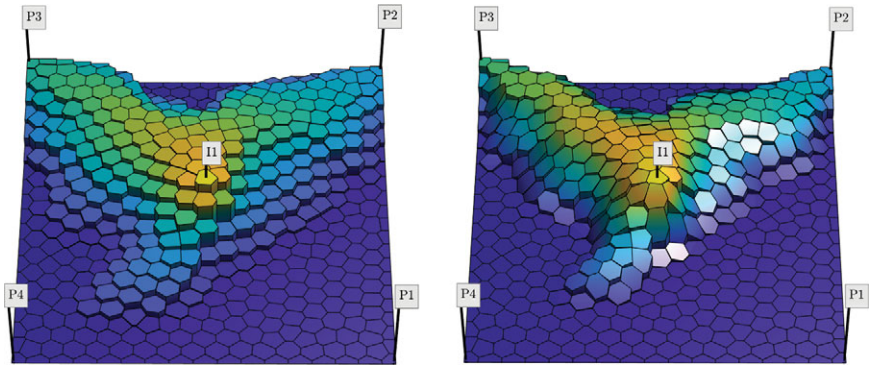


Figure 3.11 Saturations computed by the dG(0) and dG(1) schemes for the inverted five-spot case.

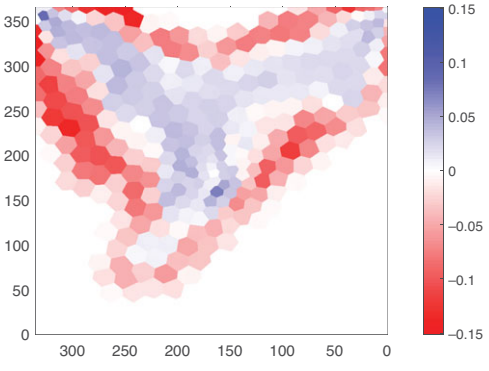


Figure 3.12 Difference in cell-averaged values for the dG(1) and the dG(0) (SPU) schemes shown in Figure 3.11. Because both solutions are mass conservative, dG(0) will have *lower* values than dG(1) behind the displacement front and *higher* values ahead of the front.

interfaces, as can clearly be seen in Figure 3.11 both for dG(0) and dG(1). From the difference in the corresponding cell-averaged values shown in Figure 3.12 it is clear that dG(0)/SPU smears the solution much more than dG(1), and this affects the prediction of water breakthrough (see Figure 3.13). An increased number of transport iterations, from 112 for dG(0) to 417 for dG(1), bears witness to the increased nonlinearity.

3.4.4 Grid-Orientation Errors for Adverse Mobility Ratios

In the standard SPU scheme, the intercell phase fluxes are computed entirely based on cell-averaged saturation values from the two grid cells on opposite sides of each

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

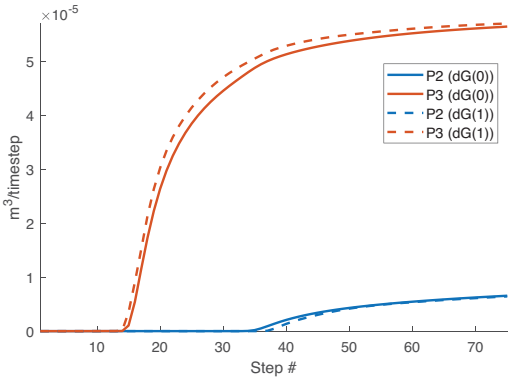


Figure 3.13 Water rate in the two producers with water breakthrough during the simulation period. Because dG(0) smears more than dG(1), it will *overestimate* the time to breakthrough in P3, which has early breakthrough because of fingering, and *underestimate* the breakthrough in P2, where the approaching displacement front is more planar.

interface. This dimension-by-dimension type of evaluation introduces preferential flow along the axial directions of the grid and can lead to strong grid-orientation effects when the displacing phase is more mobile than the displaced phase. The resulting errors grow in time as a result of the dynamic coupling between the pressure and transport equations. Higher-order dG schemes use subscale saturations inside each cell to evaluate the intercell fluxes, which significantly reduces dynamic grid-orientation errors for cases with adverse mobility ratios.

To illustrate, we revisit the setup from subsection 10.4.2 in the MRST textbook [9], comparing solutions of a two-phase, quarter-five-spot problem computed on two different Cartesian grids. The first 32×32 uniform grid follows the axial directions. This induces a preferential flow direction in the SPU scheme toward the stagnant zones along the axes connecting the producers and impedes flow along the diagonal from injector to producer. The second grid has the same resolution but follows the diagonal directions between injectors and producers. The SPU scheme will therefore exaggerate flow in the diagonal direction, as seen in the left plot of Figure 3.14.

The second-order dG(1) scheme has four degrees of freedom inside each grid cell, which effectively means that the saturation-dependent parameters are evaluated in a more multidimensional manner than in the SPU scheme. As a result, the dynamic grid-orientation errors are eliminated almost entirely and can only be seen near the leading displacement front, which is resolved much sharper on both grids. (Altogether, dG(1) required 66% and 41% more nonlinear transport iterations than SPU on the original and rotated grids, respectively.)

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

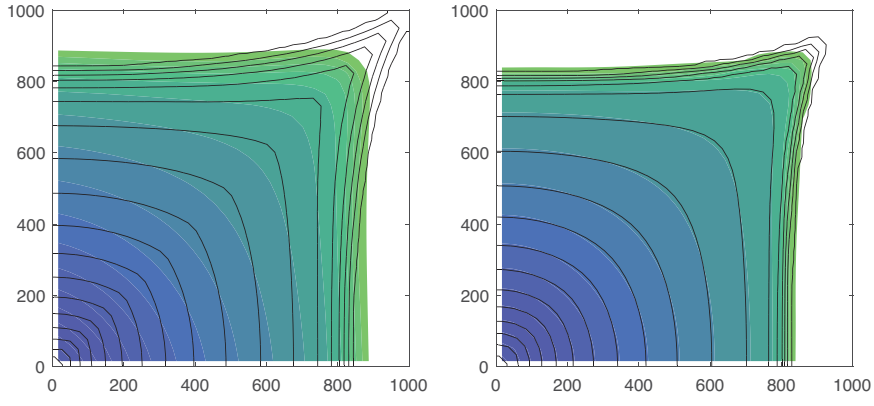


Figure 3.14 Quarter-five-spot solutions for displacement scenario with an adverse mobility ratio of 10:1 between water and oil, computed on the rotated (solid lines) and original (colors) geometry with a 32×32 grid. The left plot shows SPU and the right plot shows dG(1). (Source code: `dgExampleGridOrient`.)

3.4.5 Channelized Medium

Higher-order discretizations are particularly useful to accurately capture viscous fingers that develop in strongly heterogeneous or fractured media. To illustrate this, we consider another subsample from the SPE10 benchmark case, this time a 60×110 subset from the 51st layer, which is part of the fluvial Upper Ness formation. We move two of the producers in the original inverted five spot so that they all are completed in cells with good sand quality. We also move the injector a bit to the north to ensure a better overall sweep. Full setup of the example:

```
[state0, imodel, schedule] =  setupSPE10_AD('layers', 51, ...
           'J',1:110, 'make2D', true, 'T', 3*year, 'dt', 20*day);
G  = computeCellDimensions(imodel.G);
schedule.control.W(3).cells = 6550;
schedule.control.W(4).cells = 6578;
schedule.control.W(5).cells = 3811;
model = GenericBlackOilModel(G, imodel.rock, imodel.fluid, 'gas', false);
```

We can then construct sequential simulators with dG(0) and dG(1) exactly as in the previous example. As shown in Figure 3.15, the second-order dG(1) scheme resolves the advancing fingers more sharply and predicts a more rapid movement of the tip of the fingers compared with dG(0). Increasing the accuracy from second to third order does not improve the resolution of the advancing fingers significantly, as also observed previously by [13]. Whereas higher-order solutions primarily stay within the high-permeability channels, the dG(0) solution is more smeared out and

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

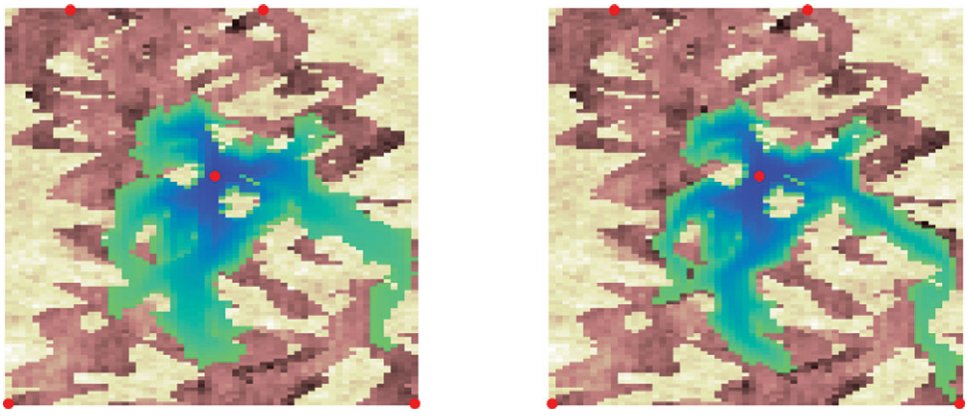


Figure 3.15 Water saturation overlain on permeability for the Upper Ness subsample simulated by dG(0) (left plot) and dG(1) (right plot). (Source code: `dgExampleNess.m`.)

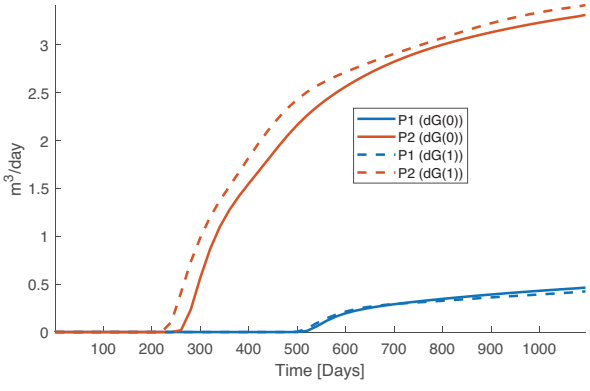


Figure 3.16 Water production rate in the two wells located along the southern perimeter of the Upper Ness test case in Figure 3.15. Well P1 is located in the southwest corner and P2 in the southeast corner. Because dG(1) introduces less smearing of the water fingers, it also predicts earlier water breakthrough.

will overestimate areal sweep and the time to water breakthrough; see Figure 3.16. (Altogether, dG(1) uses 2.9 times as many nonlinear transport iterations as the SPU scheme.)

3.5 Concluding Remarks

This chapter has introduced you to implicit discontinuous Galerkin methods, which you can use to increase the resolution of the transport step in a sequential implicit formulation. This may be particularly beneficial for cases with unfavorable

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

displacement ratios and significant fingering introduced by heterogeneity. We end the chapter by discussing some advantages and disadvantages of dG methods.

Size of the discrete system: One advantage of dG methods is that because the higher-order approximation is introduced locally to each cell, it is relatively simple to develop adaptive versions that adapt the order in space. The disadvantage is that increasing the order significantly increases the number of degrees of freedom per unknown: to 3 in 2D and 4 in 3D for dG(1), to 6 in 2D and 10 in 3D for dG(2), and so on. For compositional flow and other transport equations with many primary unknowns, this will rapidly amplify the size of the linear systems. In our experience, however, the largest effect on accuracy comes from replacing SPU by dG(1), and in 3D it may even be sufficient to only increases the order in the lateral directions, which requires fewer quadrature points than a full dG(1) scheme.

Nonlinear solvers: Increasing spatial order also increases the nonlinearity of the discrete residual equations, so that standard Newton–Raphson solvers may not be able to take as large time steps for a dG(1) method as for SPU. However, because the extra degrees of freedom are local to each cell, the nonlinear system arising from a dG(k) discretization will have the exact same sparsity structure as the SPU system, except that each matrix entry for the SPU system is replaced by an $n_{\text{dof}} \times n_{\text{dof}}$ matrix block in the dG(k) system.

In sections 5.3 and 10.3 of the MRST textbook [9] we discussed how it is possible to permute the SPU discretization matrix to a (block) triangular form (see Figure 3.17), using an ordering based on fluid potential or total flux, so that it can be solved very efficiently using a nonlinear back-substitution method that solves a sequence of single-cell problems for cases that predominantly have cocurrent flow. In most displacement scenarios, the Newton updates are localized around (strong) displacement fronts so that cells further away can converge within a few iterations or not require any iterations at all if the cell residual is below the prescribed iteration tolerance. By localizing the nonlinear iteration this way, significant computational savings are possible. The same approach can be applied to dG schemes, except that we now have to solve local $n_{\text{dof}} \times n_{\text{dof}}$ nonlinear systems, which still is significantly less expensive than having to solve the full global nonlinear problem. This advantage of dG over other high-resolution schemes was first observed by Natvig and Lie [13].

Disclaimers about the current implementation: As with many other add-on modules to MRST, the `dg` module is primarily a proof-of-concept implementation that has not yet been optimized for computational speed. The solvers in `dg` are admittedly quite slow, in part because of many redundant computations in the

https://doi.org/10.1017/9781009019781.007 Published online by Cambridge University Press

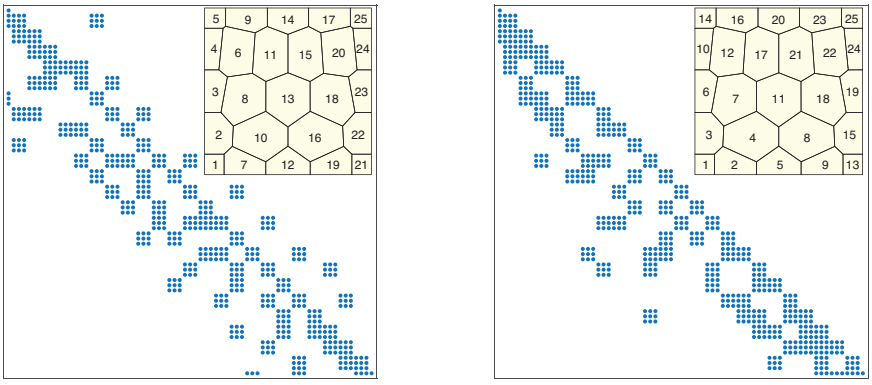


Figure 3.17 The sparsity structure of dG(1) on a perpendicular bisector grid with natural and potential ordering. An injector is placed in the southwest cell and a producer in the northeast cell; the polynomial order of dG is set to zero in both of these cells. (Source code: `dgShowSparsity.m`.)

evaluation of numerical cubatures and in part because a few key data elements are accessed row-wise, which is at odds with MATLAB's internal compressed sparse column (CSC) storage format. You should therefore try out the dG solvers on a representative smaller case before attempting to run models with more than $\mathcal{O}(10^4)$ grid cells. The standard solvers in MRST are hardly affected if you represent your domain using a mesh of higher dimensions than necessary; e.g., represent a Cartesian $n \times m$ grid as an $n \times m \times 1$ model. For dG(k) with $k > 0$, this will introduce unneeded basis functions and cubature points and incur extra computational cost. As a general precaution, you may consider shortening the time step somewhat compared to the SPU scheme to avoid potential convergence problems in the nonlinear solver. (We also remark that the dG(2) solver is less robust and does not seem to provide much more resolution than dG(1) due to the implicit time discretization.)

Implementing a finite element–type method like dG for unstructured polytopal grids is significantly more involved than a finite-volume method. It requires an appropriate set of basis functions and accompanying cubature rules for numerical integration. Constructing efficient cubature rules is challenging for polytopal grids. Robust and correct limiting strategies are also not straightforward to implement for implicit simulations. Herein, we have presented simple choices that seem to work reasonably well for dG(1) applied to immiscible multiphase flow. Optimizing and tuning these choices and applying them to more complex flow scenarios will be subject to further research.

Acknowledgement. The research leading up to the `dg` module has been partially funded by the Research Council of Norway through grant no. 244361 and by Total E&P Norway.

References

[1] R. Cools. An encyclopaedia of cubature formulas. *Journal of Complexity*, 19(3): 445–453, 2003. doi: 10.1016/S0885-064X(03)00011-6.

[2] A. Dedner, B. Kane, R. Klöfkorn, and M. Nolte. Python framework for hp-adaptive discontinuous Galerkin methods for two-phase flow in porous media. *Applied Mathematical Modelling*, 67:179–200, 2019. doi: 10.1016/j.apm.2018.10.013.

[3] F. P. Hamon and B. T. Mallison. Fully implicit multidimensional hybrid upwind scheme for coupled flow and transport. *Computer Methods in Applied Mechanics and Engineering*, 358:112606, 2020. doi: 10.1016/j.cma.2019.112606.

[4] A. Harten. High resolution schemes for hyperbolic conservation laws. *Journal of Computational Physics*, 135(2):260–278, 1997. doi: 10.1006/jcph.1997.5713. Reprinted from Volume 49, Number 3, March 1983, pp. 357–393.

[5] J. S. Hesthaven and T. Warburton. *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications*. Number 54 in *Texts in Applied Mathematics*. Springer Science & Business Media, New York, 2008. doi: 10.1007/978-0-387-72067-8.

[6] G.-S. Jiang and C.-W. Shu. Efficient implementation of weighted ENO schemes. *Journal of Computational Physics*, 126(1):202–228, 1996. doi: 10.1006/jcph.1996.0130.

[7] E. Keilegavlen, J. E. Kozdon, and B. T. Mallison. Multidimensional upstream weighting for multiphase transport on general grids. *Computational Geosciences*, 16(4):1021–1042, 2012. doi: 10.1007/s10596-012-9301-7.

[8] R. A. Klausen, A. F. Rasmussen, and A. F. Stephansen. Velocity interpolation and streamline tracing on irregular geometries. *Computational Geosciences*, 16(2): 261–276, 2012. doi: 10.1007/s10596-011-9256-0.

[9] K.-A. Lie. *An Introduction to Reservoir Simulation Using MATLAB/GNU Octave: User Guide for the MATLAB Reservoir Simulation Toolbox (MRST)*. Cambridge University Press, Cambridge, UK, 2019. doi: 10.1017/9781108591416.

[10] K.-A. Lie, T. S. Mykkeltvedt, and O. Møyner. A fully implicit WENO scheme on stratigraphic and unstructured polyhedral grids. *Computational Geosciences*, (24):405–423, 2020. doi: 10.1007/s10596-019-9829-x.

[11] X.-D. Liu, S. Osher, T. Chan, et al. Weighted essentially non-oscillatory schemes. *Journal of Computational Physics*, 115(1):200–212, 1994. doi: 10.1006/jcph.1994.1187.

[12] B. Müller, F. Kummer, and M. Oberlack. Highly accurate surface and volume integration on implicit domains by means of moment-fitting. *International Journal for Numerical Methods in Engineering*, 96(8):512–528, 2013. doi: 10.1002/nme.4569.

[13] J. R. Natvig and K.-A. Lie. Fast computation of multiphase flow in porous media by implicit discontinuous Galerkin schemes with optimal ordering of elements. *Journal of Computational Physics*, 227(24):10108–10124, 2008. doi: 10.1016/j.jcp.2008.08.024.

[14] J. Qiu and C.-W. Shu. Runge–Kutta discontinuous Galerkin method using WENO limiters. *SIAM Journal on Scientific Computing*, 26(3):907–929, 2005. doi: 10.1137/S1064827503425298.

[15] W. H. Reed and T. R. Hill. Triangular mesh methods for the neutron transport equation. Technical Report LA-UR-73-479, Los Alamos Scientific Laboratory, Los Alamos, NM, 1973.

[16] B. Riviere. *Discontinuous Galerkin Methods for Solving Elliptic and Parabolic Equations: Theory and Implementation*. SIAM, Philadelphia, 2008. doi:10.1137/1.9780898717440.

[17] C.-W. Shu. TVB uniformly high-order schemes for conservation laws. *Mathematics of Computation*, 49(179):105–121, 1987. doi: 10.1090/S0025-5718-1987-0890256-5.
[18] P. Solin, K. Segeth, and I. Dolezel. *Higher-Order Finite Element Methods*. Chapman and Hall/CRC, New York, 2003. doi: 10.1201/9780203488041.
[19] H. C. Yee and A. Harten. Implicit TVD schemes for hyperbolic conservation laws in curvilinearcoordinates. *AIAA Journal*, 25(2):266–274, 1987. doi: 10.2514/3.9617.
[20] J. Zhu, J. Qiu, C.-W. Shu, and M. Dumbser. Runge–Kutta discontinuous Galerkin method using WENO limiters II: unstructured meshes. *Journal of Computational Physics*, 227(9):4330–4353, 2008. doi: 10.1016/j.jcp.2007.12.024.