Archive of Numerical Software
vol. 5, no. 1, 2017, pages 111–128
DOI: 10.11588/ans.2017.1.27719

The Dune-DPG library for solving PDEs with Discontinuous Petrov–Galerkin finite elements

Felix Gruber[1], Angela Klewinghaus[1], and Olga Mula[2]

[1]IGPM, RWTH Aachen, Templergraben 55, 52056 Aachen, Germany
[2]Université Paris-Dauphine, PSL Research University, CNRS, UMR 7534, CEREMADE, 75016 Paris, France

Received: February 7th, 2016; **final revision:** November 7th, 2016; **published:** March 6th, 2017.

Abstract: In the numerical solution of partial differential equations (PDEs), a central question is the one of building variational formulations that are inf-sup stable not only at the infinite-dimensional level, but also at the finite-dimensional one. These properties are important since they represent the rigorous foundations for a posteriori error control and the development of adaptive strategies. The essential difficulty lies in finding *systematic* procedures to build variational formulations for which these desirable stability properties are (i) provable at the theoretical level while (ii) the approach remains implementable in practice and (iii) its computational complexity does not explode with the problem size. In this framework, the so-called Discontinuous Petrov–Galerkin (DPG) concept seems a promising approach to enlarge the scope of problems beyond second order elliptic PDEs for which this is possible. In the context of DPG, the result for the elliptic case was proven by Gopalakrishnan and Qiu [2014] and requires a p-enriched test space. Recently, the same type of result has been proven by Broersen et al. [2015] for certain classes of linear transport problems using an appropriate hp-enrichment to build the finite dimensional test space. In the light of this new result, we present Dune-DPG, a C++ library which allows to implement the test spaces introduced in Broersen et al. [2015]. The library is built upon the multi-purpose finite element package Dune (see Blatt et al. [2016]). In this paper, we present the current version 0.2.1 of Dune-DPG which has so far been tested only for elliptic and transport problems. An example of use via a simple transport equation is described. We conclude outlining future work and applications to more complex problems. Dune-DPG is licensed under the GPL 2 with runtime exception and a source code tarball is available together with this paper.

1 Introduction

General context: Let Ω be a domain of $\mathbb{R}^d$ ($d \geq 1$) and $\mathbb{U}$, $\mathbb{V}$ two Hilbert spaces defined over Ω and endowed with norms $\|\cdot\|_{\mathbb{U}}$ and $\|\cdot\|_{\mathbb{V}}$, respectively. The normed dual of $\mathbb{V}$, denoted $\mathbb{V}'$, is endowed with the norm

$$\|\ell\|_{\mathbb{V}'} := \sup_{v\in\mathbb{V}} \frac{|\ell(v)|}{\|v\|_{\mathbb{V}}}, \quad \forall \ell \in \mathbb{V}'.$$

Let $\mathcal{B}\colon \mathbb{U} \to \mathbb{V}$ be a boundedly invertible linear operator and let $b\colon \mathbb{U} \times \mathbb{V} \to \mathbb{R}$ be its associated continuous bilinear form defined by $b(w, v) = (\mathcal{B}w)(v)$, $\forall (w, v) \in \mathbb{U} \times \mathbb{V}$. We consider the operator equation

$$\begin{aligned} &\text{Given } f \in \mathbb{V}', \text{ find } u \in \mathbb{U} \text{ s. t.} \\ &\mathcal{B}u = f, \end{aligned} \tag{1}$$

or, equivalently, the variational problem

$$\begin{aligned} &\text{Given } f \in \mathbb{V}', \text{ find } u \in \mathbb{U} \text{ s. t.} \\ &b(u, v) = f(v), \quad \forall v \in \mathbb{V}. \end{aligned} \tag{2}$$

Let $0 < \gamma \leq 1$ be a lower bound for the (infinite-dimensional) inf-sup constant

$$\inf_{w \in \mathbb{U}} \sup_{v \in \mathbb{V}} \frac{b(w, v)}{\|w\|_{\mathbb{U}} \|v\|_{\mathbb{V}}} \geq \gamma > 0.$$

Since $\mathcal{B}$ is invertible, problem (1) admits a unique solution $u \in \mathbb{U}$ and for any approximation $\bar{u} \in \mathbb{U}$ of u,

$$\|\mathcal{B}\|^{-1}_{L(\mathbb{U},\mathbb{V}')} \|f - \mathcal{B}\bar{u}\|_{\mathbb{V}'} \leq \|u - \bar{u}\|_{\mathbb{U}} \leq \gamma^{-1} \|f - \mathcal{B}\bar{u}\|_{\mathbb{V}'}. \tag{3}$$

From (3), it follows that the error $\|u - \bar{u}\|_{\mathbb{U}}$ is equivalent to the residual $\|f - \mathcal{B}\bar{u}\|_{\mathbb{V}'}$. The residual contains known quantities and its estimation on an appropriate finite-dimensional space opens the door to rigorously founded a posteriori concepts. However, note that the information that the estimator can give is only meaningful when the variational formulation is well-conditioned, i. e., for $\|\mathcal{B}\|_{L(\mathbb{U},\mathbb{V}')}$ and γ being as close to one as possible. Assuming that we have this property of well-conditioning, a crucial point is that this needs to be inherited at the finite-dimensional level. This issue has been well explored for standard Galerkin methods (i. e. when $\mathbb{U} = \mathbb{V}$) and allows to appropriately address most parabolic and second order elliptic problems with a wide variety of finite element methods. However, there are problems for which standard Galerkin methods do not lead to a stable discretization (one example being transport-dominated PDEs). In this context, one can move to a Petrov-Galerkin framework where the main guiding ideas are:

- It is possible to find a test space $\mathbb{V}$ and a norm for $\mathbb{V}$ which yield $\gamma = 1$. $\mathbb{V}$ and its norm depend on the specific PDE and might not coincide with $\mathbb{U}$.
- For a given finite-dimensional trial space $\mathbb{U}_H$, there exists a corresponding optimal test space $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ such that the discrete inf-sup condition

$$\inf_{w_H \in \mathbb{U}_H} \sup_{v \in \mathbb{V}^{\text{opt}}(\mathbb{U}_H)} \frac{b(w_H, v)}{\|w_H\|_{\mathbb{U}} \|v\|_{\mathbb{V}}} \geq \gamma$$

  holds with the same constant γ as the infinite-dimensional one. For more details on this, see Section 2.1.

Since, in general, even the approximate computation of the optimal test space requires the solution of global problems, there are essentially two ways to make the computation affordable. One is the introduction of a mixed formulation which avoids the computation of the optimal test spaces and only uses them indirectly. This approach was used in Dahmen et al. [2012] to construct a general adaptive scheme when $\mathbb{U} = L_2(\Omega)$ and to show convergence under certain abstract conditions (which have to be verified for concrete applications). It was also employed in the context of reduced-basis construction for transport-dominated problems (see Dahmen et al. [2014]).

The other way is to make computations affordable by localization so that the optimal test spaces can be computed by solving local problems. This is the approach taken in the DPG methodology, initiated and developed mainly by L. Demkowicz and J. Gopalakrishnan (see e. g. Demkowicz

and Gopalakrishnan [2011], Gopalakrishnan and Qiu [2014], Demkowicz and Gopalakrishnan [2015]). Since, in general, the local problems to find $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ are still infinite dimensional, the DPG method approximates the exact optimal test functions in the context of a discontinuous Petrov–Galerkin formulation. As a result, $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ is replaced in practical computations by a finite dimensional space called "near-optimal test-space", which we will denote $\mathbb{V}^{\text{n.opt}}$. This is the space which is eventually used in the solution of the discrete problem. For fairly broad classes of PDEs, there exist abstract results showing that the inf-sup stability of the discrete problem with $\mathbb{V}^{\text{n.opt}}$ is preserved provided that $\mathbb{V}^{\text{n.opt}}$ is close enough to the true optimal test-space $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ in a certain abstract sense. We refer to Roberts et al. [2014] and also the concept of delta proximality introduced in Dahmen et al. [2012] for results in this respect. Deriving more quantifiable conditions on the form of $\mathbb{V}^{\text{n.opt}}$ is a more involved task which depends on the PDE. It is nevertheless important since it helps to quantify the complexity of the approach in more specific terms. This type of result was first proven for second order elliptic problems by Gopalakrishnan and Qiu [2014]. It requires a p-enrichment strategy (with respect to the polynomial degree of $\mathbb{U}_H$) to build $\mathbb{V}^{\text{n.opt}}$. Recently, the same type of result has been derived by Broersen et al. [2015] for certain classes of linear transport problems. It is proven there that the use of a certain hp-enrichment in the construction of $\mathbb{V}^{\text{n.opt}}$ guarantees a sufficiently good near-optimal test space. Beyond these theoretically backed-up cases, we also note that numerical evidence illustrates the good stability properties of DPG for a larger spectrum of PDEs than elliptic and transport problems. Without being exhaustive, we can find works on convection–diffusion [Broersen and Stevenson, 2015, Niemi et al., 2013], elasticity and Stokes [Carstensen et al., 2014, Roberts et al., 2014], Maxwell equations [Carstensen et al., 2016] and the Helmholtz equation [Demkowicz et al., 2012]. Also, we note that the idea of hp-enrichment to build $\mathbb{V}^{\text{n.opt}}$ was explored in numerical experiments about convection-diffusion in Niemi et al. [2013].

Motivations and contributions of the paper: In this paper, we explain the construction of the Dune-DPG library which aims at solving different types of PDEs with a DPG variational formulation. The main guidelines for its construction have been:

- to give the user as much liberty as possible in the nature of the problem to be addressed, in the nature of the test and trial spaces to be used and in the geometry and mesh refinement,
- to endow the code with a modular structure in order to ensure an easy access to low-level functionalities for which a fine control is required while keeping a high-level view for the rest of the code.

In the light of the recent results by Broersen et al. [2015] and in order to have a rigorous back-up on the stability of the calculations for the largest possible scope of problems, a special emphasis has been put on giving the possibility to use h and p refined spaces for the construction of the near-optimal test space $\mathbb{V}^{\text{n.opt}}$. This point is actually the main novelty that the present library offers with respect to other existing DPG libraries like *Camellia* (see Roberts [2014]). It is nevertheless significant in the sense that this capability seems well suited to investigate novel numerical schemes with rigorous error bounds for more elaborate transport based PDEs such as kinetic problems. A work in this direction is currently ongoing and will be subject of a future publication.

Dune-DPG is licensed under the GPL 2 with runtime exception and its latest version (0.2.1) is released together with this paper[1]. The library is based upon the multi-purpose finite element package Dune (see, e. g., Bastian et al. [2008]). In particular, we use version 2.4.1 [Blatt et al., 2016] of the core modules Dune-Grid [Bastian et al., 2008], Dune-ISTL [Blatt and Bastian, 2006], Dune-Geometry, Dune-Functions [Engwer et al., 2015], Dune-LocalFunctions and Dune-TypeTree. As a consequence, for the users of Dune, the library represents a relatively simple way to experiment with basic DPG concepts.

[1] The whole Git history of Dune-DPG can be found at `https://gitlab.dune-project.org/felix.gruber/dune-dpg`. We try to keep the master branch compatible to the current development branches of the Dune core modules.

Finally, we would like to point out that Dune-DPG is still under active development and the present release comes with a couple of limitations which will be fixed in future versions. These are:

- the current implementation is limited to problems with constant coefficients,
- parallelization of the code has not been explored,
- the library has been tested so far in elliptic and pure transport problems but, as already brought up, it is currently being used to study more involved PDEs.

Layout of the paper: To show how the library works, the paper is organized as follows: In Section 2, we summarize the mathematical concepts of DPG that are relevant to understand the library. As an example, we explain at the end of this section how the ideas can be applied to a simple transport problem following the theory of Broersen et al. [2015]. Then, in Section 3, we present the different building blocks that form the library. We explain how they interact and how they make use of some features of the Dune framework upon which our library is built. Finally, in Section 4, we validate Dune-DPG by giving concrete results related to the solution of a simple transport problem. Some performance results are also given.

2 Theoretical Foundations for DPG

As already brought up in the introduction, the DPG concept was initiated and developed mainly by L. Demkowicz and J. Gopalakrishnan (see e. g. Demkowicz and Gopalakrishnan [2011], Gopalakrishnan and Qiu [2014]). Other relevant results concerning theoretical foundations are Broersen and Stevenson [2014, 2015] and, more recently, Broersen et al. [2015]. The strategy followed in DPG to contrive stable variational formulations is based on the concept of *optimal test spaces* and their practical approximation through the solution of *local* problems in the context of a discontinuous Petrov–Galerkin variational formulation. The two following sections explain more in detail these two fundamental ideas.

2.1 The concepts of optimal and near-optimal test spaces

Assuming that we start from a well-posed and well-conditioned infinite-dimensional variational formulation (2), we look for a formulation at the finite-dimensional level which inherits these desirable features. Let $H > 0$ be a parameter (H will later be associated to the size of a mesh Ω_H of Ω). For any given finite-dimensional trial space $\mathbb{U}_H$ of dimension $\mathcal{N}$ (that depends on H), there exists a so-called optimal test space $\mathbb{V}^{\text{opt}}(\mathbb{U}_H)$ of the same dimension. It is called optimal because the finite-dimensional version of problem (2),

$$\begin{aligned} &\text{Find } u_H \in \mathbb{U}_H \text{ s. t.} \\ &b(u_H, v) = f(v), \quad \forall v \in \mathbb{V}^{\text{opt}}(\mathbb{U}_H), \end{aligned} \tag{4}$$

is well posed and

$$\inf_{w_H \in \mathbb{U}_H} \sup_{v \in \mathbb{V}^{\text{opt}}(\mathbb{U}_H)} \frac{b(w_H, v)}{\|w_H\|_{\mathbb{U}} \|v\|_{\mathbb{V}}} \geq \gamma.$$

In other words, the discrete inf-sup condition is bounded with the same constant γ that is involved in the infinite-dimensional problem. This implies that the discrete problem has the same stability properties as the infinite-dimensional problem. Therefore the residual $\|f - \mathcal{B}u_H\|_{\mathbb{V}'}$ is equivalent to the actual error $\|u - u_H\|_{\mathbb{U}}$ with the same constants exhibited in (3). Since these constants do not depend on H, $\|f - \mathcal{B}u_H\|_{\mathbb{V}'}$ is a *robust* error bound that is suitable for adaptivity since we can decrease H without degrading the constants of equivalence.

Unfortunately, the optimal test space $\mathbb{V}^{\mathrm{opt}}(\mathbb{U}_H)$ is not computable in practice. Indeed, if $\{u_H^i\}_{i=1}^{\mathcal{N}}$ spans a basis of $\mathbb{U}_H$, then the set of functions $\{v^i\}_{i=1}^{\mathcal{N}}$ defined through the variational problems,

$$i \in \{1, \dots, \mathcal{N}\}, \quad \langle v^i, v\rangle_{\mathbb{V}} = b(u_H^i, v), \quad \forall v \in \mathbb{V} \tag{5}$$

spans a basis of $\mathbb{V}^{\mathrm{opt}}(\mathbb{U}_H)$. Since these problems are formulated in the infinite-dimensional space $\mathbb{V}$, they cannot be computed exactly (in addition, the problems are global). To address this issue, problems (5) are $\mathbb{V}$-projected to a finite-dimensional subspace $\mathbb{V}_h$ that will be called test-search space. Therefore, in practice, an approximation $\{\bar{v}^i\}_{i=1}^{\mathcal{N}}$ to the set of functions $\{v^i\}_{i=1}^{\mathcal{N}}$ is computed by solving for all $i \in \{1, \dots, \mathcal{N}\}$,

$$\langle \bar{v}^i, v_h\rangle_{\mathbb{V}} = b(u_H^i, v_h), \quad \forall v_h \in \mathbb{V}_h. \tag{6}$$

This defines a projected test space $\mathbb{V}^{\mathrm{n.opt}}(\mathbb{U}_H, \mathbb{V}_h) := \mathrm{span}\{\bar{v}^i\}_{i=1}^{\mathcal{N}}$. For the elliptic case and some classes of transport problems, it has been shown that it is possible to exhibit test-search spaces $\mathbb{V}_h$ (which depend on the initial $\mathbb{U}_H$) such that $\mathbb{V}^{\mathrm{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ is close enough to the optimal $\mathbb{V}^{\mathrm{opt}}(\mathbb{U}_H)$ to allow that the discrete inf-sup constant

$$\gamma_H := \inf_{u_H \in \mathbb{U}_H} \sup_{\bar{v} \in \mathbb{V}^{\mathrm{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)} \frac{b(u_H, \bar{v})}{\|u_H\|_{\mathbb{U}_H} \|\bar{v}\|_{\mathbb{V}_h}} \tag{7}$$

is bounded away from 0 uniformly in H. For this reason, $\mathbb{V}^{\mathrm{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ is called a near-optimal test-space. In the case of transport problems, the recent work of Broersen et al. [2015] shows that good test-search spaces $\mathbb{V}_h$ can be found when they are defined over a refinement Ω_h of Ω_H.

The near-optimal test space $\mathbb{V}^{\mathrm{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ is the one that is computed in practice in the Dune-DPG library. The finite-dimensional variational formulation that is eventually solved reads

$$\begin{aligned} &\text{Find } u_H \in \mathbb{U}_H \text{ s. t.} \\ &b(u_H, \bar{v}) = f(\bar{v}), \quad \forall \bar{v} \in \mathbb{V}^{\mathrm{n.opt}}(\mathbb{U}_H, \mathbb{V}_h). \end{aligned} \tag{8}$$

It can be expressed as a linear system of the form $Ax = F$, $A \in \mathbb{R}^{\mathcal{N}\times\mathcal{N}}$, $x \in \mathbb{R}^{\mathcal{N}}$, $F \in \mathbb{R}^{\mathcal{N}}$. It can be proven that A is by construction symmetric positive definite. The assembly of the system and its solution in Dune-DPG are explained in Section 3.1.

2.2 The concept of localization

Depending on the choice of $\mathbb{V}$ and $\mathbb{V}_h$, the solution of (6) to derive the near-optimal basis functions of $\mathbb{V}^{\mathrm{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ might be costly. This is because these $\mathcal{N}$ problems are, in general, global in the whole domain Ω and they cannot be decomposed into local ones. Furthermore, if the resulting near-optimal basis functions have global support, the solution of the finite-dimensional variational problem (8) is costly as well because the resulting system matrix A is full.

To prevent this, we need an appropriate variational formulation with a well-chosen test space $\mathbb{V}$ which has a product structure on the coarse grid Ω_H,

$$\mathbb{V} := \prod_{K \in \Omega_H} \mathbb{V}_K, \tag{9}$$

where $\mathrm{supp}(v) \subset K$ for any $v \in \mathbb{V}_K$. In particular, the restriction of the $\mathbb{V}$-scalar product to $K \in \Omega_H$ has to be a scalar product for $\mathbb{V}_K$:

$$\langle \cdot, \cdot \rangle_{\mathbb{V}}|_K = \langle \cdot, \cdot \rangle_{\mathbb{V}_K}$$

The test-search space $\mathbb{V}_h$ will be chosen in such a way, that it has the same product structure as $\mathbb{V}$,

$$\mathbb{V}_h := \prod_{K \in \Omega_H} \mathbb{V}_{h,K}, \quad \mathbb{V}_{h,K} \subset \mathbb{V}_K.$$

Therefore, for any $1 \leq i \leq \mathcal{N}$, the near-optimal test function $\bar{v}^i$ can be written as

$$\bar{v}^i = \sum_{K \in \Omega_H} \bar{v}^i_K \chi_K,$$

where χ_K is the characteristic function of cell K. Additionally, we need a decomposition of the bilinear form as a sum over mesh cells of Ω_H,

$$b\,(u, v) = \sum_{K \in \Omega_H} b_K\,(u, v)\,, \quad \forall v \in \prod_{K \in \Omega_H} \mathbb{V}_K. \tag{10}$$

Then, for every $K \in \Omega_H$, $\bar{v}^i_K$ is the solution of a local problem in K,

$$\langle \bar{v}^i_K, v_{h,K} \rangle_{\mathbb{V}_K} = b_K \left(u^i_H, v_{h,K} \right), \; \forall v_{h,K} \in \mathbb{V}_{h,K}, \tag{11}$$

where $\{u^i_H\}_{i=1}^{\mathcal{N}}$ is a basis of $\mathbb{U}_H$. Therefore, finding $\bar{v}^i$ can be decomposed into a sum of problems, each one of which is localized on a mesh cell $K \in \Omega_H$. Moreover, if the support of u^i_H is included in some cell $K \in \Omega_H$, then the support of its corresponding near-optimal test function $\bar{v}^i$ is also a subset of K (and the neighboring cells in some cases). In other words, we would have $\bar{v}^i = \bar{v}^i_K \chi_K$ or $\bar{v}^i = \sum_{K' \in \text{ neigh(K)}} \bar{v}^i_{K'} \chi_{K'}$. Hence, if the basis functions u^i_H of $\mathbb{U}_H$ have local support, the resulting system matrix A is sparse.

2.3 An example: a linear transport equation

Let $\Omega = (0, 1)^2$ and β be a vector of $\mathbb{R}^2$ with norm one. For any $x \in \partial\Omega$, let $n(x)$ be its associated outer normal vector. Then

$$\Gamma_- := \{x \in \partial\Omega \mid \beta \cdot n(x) < 0\} \subset \partial\Omega \tag{12}$$

is the inflow-boundary for the given constant transport direction β. Given $c \in \mathbb{R}$ and a function $f \colon \Omega \to \mathbb{R}$, we consider the problem of finding the solution $\varphi \colon \Omega \to \mathbb{R}$ to the simple transport equation

$$\begin{aligned} \beta \cdot \nabla\varphi + c\varphi &= f, \quad \text{in } \Omega,\\ \varphi &= 0, \quad \text{on } \Gamma_-. \end{aligned} \tag{13}$$

If we apply the DPG approach introduced in Broersen et al. [2015] to solve this problem, we first need to introduce the following spaces. Denoting by ∇_H the piecewise gradient operator, let

$$H(\beta, \Omega_H) := \{v \in L_2(\Omega) \mid \beta \cdot \nabla_H v \in L_2(\Omega)\},$$

equipped with squared "broken" norm $\|v\|^2_{H(\beta,\Omega_H)} = \|v\|^2_{L_2(\Omega)} + \|\beta \cdot \nabla_H v\|^2_{L_2(\Omega)}$. Let also

$$H_{0,\Gamma_-}(\beta, \Omega) := \text{clos}_{H(\beta,\Omega)}\{u \in H(\beta, \Omega) \cap C(\bar{\Omega}) \mid u = 0 \text{ on } \Gamma_-\}$$

and

$$H_{0,\Gamma_-}(\beta, \partial\Omega_H) := \{w|_{\partial\Omega_H} \mid w \in H_{0,\Gamma_-}(\beta, \Omega)\}$$

equipped with quotient norm

$$\|\theta\|_{H_{0,\Gamma_-}(\beta,\partial\Omega_H)} := \inf\{\|w\|_{H(\beta,\Omega)} \mid \theta = w|_{\partial\Omega_H},\ w \in H_{0,\Gamma_-}(\beta, \Omega)\}.$$

The variational formulation reads

$$\begin{aligned} &\text{For } \mathbb{U} := L^2(\Omega) \times H_{0,\Gamma_-}(\beta, \partial\Omega_H) \text{ and } \mathbb{V} := H(\beta, \Omega_H),\\ &\text{given } f \in H(\beta, \Omega_H)', \text{ find } u := (\varphi, \theta) \in \mathbb{U} \text{ such that}\\ &b(u, v) = f(v), \quad \forall v \in \mathbb{V}. \end{aligned} \tag{14}$$

In this formulation (usually called *ultra-weak* formulation) the bilinear form $b(u, v)$ is defined by

$$b(u, v) = b\left((\varphi, \theta), v\right) = \int_{\Omega} (-\beta \cdot \nabla v \varphi + cv\varphi) \; \mathrm{d}x + \int_{\partial\Omega_H} [\![v\beta]\!]\theta \; \mathrm{d}s. \tag{15}$$

Note that this variational formulation depends on the mesh Ω_H. Also, note the presence of an additional unknown θ that lives on the skeleton $\partial\Omega_H$ of the mesh. For smooth solutions, θ agrees with the traces of φ on $\partial\Omega_H$ (i. e. the union of cell interfaces of Ω_H).

For the discretization, we replace θ by a lifting $w \in H_{0,\Gamma_-}(\beta, \Omega)$ (for details see Broersen et al. [2015]) and take for some $m \in \mathbb{N}$,

$$\mathbb{U}_H := \left(\prod_{K \in \Omega_H} \mathbb{P}_{m-1}(K) \right) \times \left(H_{0,\Gamma_-}(\beta, \Omega) \cap \prod_{K \in \Omega_H} \mathbb{P}_m(K) \right) \Bigg|_{\partial\Omega_h}, \tag{16}$$

where $\mathbb{P}_m(K)$ is the space of polynomials of degree m. A viable test-search space can be taken simply as discontinuous piecewise polynomials of slightly higher degree on the finer mesh Ω_h of Ω_H, namely

$$\mathbb{V}_h := \prod_{K \in \Omega_h} \mathbb{P}_{m+1}(K). \tag{17}$$

As a result, the discrete version of (14) reads

$$\begin{aligned} &\text{Find } u_H := (\varphi_H, w_H) \in \mathbb{U}_H \text{ such that} \\ &\tilde{b}(u_H, \bar{v}) = f(\bar{v}), \quad \forall \bar{v} \in \mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h). \end{aligned} \tag{18}$$

The bilinear form $\tilde{b}$ is slightly different from b. It reads

$$\tilde{b}(u, v) = \tilde{b}\left((\varphi, w), v\right) = \int_{\Omega} (-\beta \cdot \nabla v \varphi + cv\varphi) \; \mathrm{d}x + \int_{\partial\Omega_h} [\![v\beta]\!]w \; \mathrm{d}s. \tag{19}$$

The trace integral is over $\partial\Omega_h$ and not $\partial\Omega_H$ since $\mathbb{V}_h$ is now a broken space with respect to the finer mesh Ω_h. This is why we need a lifting w instead of θ here. Note that depending on the polynomial degree m and the refinement level of Ω_h, there might be undefined degrees of freedom of w. In this case, they have to be fixed for example by minimizing $\|w\|_{H(\beta,\Omega)}$.

3 An Overview of the Architecture of Dune-DPG

In this section we describe how the DPG method presented in Section 2 has been implemented in Dune-DPG. As already brought up, the library has been built upon the finite element package Dune.

The user starts by choosing the appropriate test-search space $\mathbb{V}_h$ and trial space $\mathbb{U}_H$ for his problem. Then, the bilinear form $b(\cdot, \cdot)$ and the inner product $\langle \cdot, \cdot \rangle_{\mathbb{V}}$ are declared via the classes `BilinearForm` and `InnerProduct` (see Section 3.1.2). They both consist of an arbitrary number of elements of the type `IntegralTerm` (see Section 3.1.3). The `DPGSystemAssembler` class handles the automatic assembly of the linear system $Ax = F$ associated to problem (8). As Section 3.1.1 explains, it includes:

- the assembly of the matrix A and the right hand side F
- the treatment of boundary conditions

To assemble A, we use the decomposition (10) and define the local matrices A_K by

$$(A_K)_{i,j} = b_K(u^j_{H,K}, \bar{v}^i_K) \tag{20}$$

where $\{u^i_{H,K}\}_{i=1}^{N_K}$ is a basis for the restriction of $\mathbb{U}_H$ to K and $\{\bar{v}^i_K\}_{i=1}^{N_K}$ is a basis for the restriction of the near-optimal test space $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ to K. To compute these local matrices, we follow the same lines as the Camellia library [Roberts, 2014] and start from a basis $\{v^j_{h,K}\}_{j=1}^{\mathcal{M}_K}$ of the test-search space $\mathbb{V}_h$ restricted to K. Then, we use the bilinear form $b(\cdot,\cdot)$ and the inner product $\langle\cdot,\cdot\rangle_{\mathbb{V}}$ to construct the matrices B_K and G_K defined by

$$(B_K)_{i,j} = b_K(u^j_{H,K}, v^i_{h,K}), \qquad (G_K)_{i,j} = \langle v^i_{h,K}, v^j_{h,K}\rangle_{\mathbb{V}}\,. \tag{21}$$

It follows from (11) that the coefficients $c^{i,j}_K$ of the basis functions $\bar{v}^i_K = \sum_{j=1}^{\mathcal{M}_K} c^{i,j}_K v^j_{h,K}$ of the near-optimal test space restricted to K are the columns of the matrix

$$C_K \coloneqq G_K^{-1} B_K \tag{22}$$

and the local matrices satisfy

$$A_K = A_K^T = B_K^T C_K. \tag{23}$$

The matrices C_K and A_K are computed by the class `TestspaceCoefficientMatrix` which also offers a feature to store those matrices to reduce computational costs in case of constant coefficients (see Section 3.1.4). Finally, the `DPGSystemAssembler` class constructs the global matrix A out of the local matrices A_K. In addition to these classes, the class `ErrorTools` handles the computation of a posteriori estimators following the guidelines that are given in Section 3.2.

3.1 Assembling the discrete system for a given PDE

The following subsections describe the `DPGSystemAssembler` class and all the classes used by it.

3.1.1 DPGSystemAssembler The assembly of the discrete system $Ax = F$ derived from the variational problem is handled by the class `DPGSystemAssembler`. Before we can create a `DPGSystemAssembler` by calling the method `make_DPGSystemAssembler` we first have to define the trial space $\mathbb{U}_H$, the near-optimal test space $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ as well as the bilinear form b and the inner product that describe our DPG system. The bilinear form is an object of the class `BilinearForm` which is explained in Section 3.1.2. As for the spaces $\mathbb{U}_H$ and $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$, they are both given by a `std::tuple` composed of (scalar) global basis functions from DUNE-FUNCTIONS. The reason to use a tuple is to handle problems involving several unknowns. For instance, in the ultra-weak formulation introduced for the transport problem in Section 2.3, we have two unknowns (φ, w) and $\mathbb{U}_H$ is a product of two spaces.

The following lines of code taken from `src/plot_solution.cc` give an overview of how to set up the transport problem from Section 2.3. The individual steps will be explained in the following subsections. The example uses UG (see Bastian et al. [1997]) to represent the mesh but the code admits other grid implementations. We refer to the `README` file included in the library for details on this and on how to launch the program.

src/plot_solution.cc

```
128   using FEBasisInterior = Functions::LagrangeDGBasis<GridView, 1>;
129   FEBasisInterior spacePhi(gridView);
130
131   using FEBasisTraceLifting = Functions::PQkNodalBasis<GridView, 2>;
132   FEBasisTraceLifting spaceW(gridView);
133
134   auto solutionSpaces = std::make_tuple(spacePhi, spaceW);
135
```

```
136     using FEBasisTest
137         = Functions::PQkDGRefinedDGBasis<GridView, 1, 3>;
138     auto testSearchSpaces = std::make_tuple(FEBasisTest(gridView));
139 
140     auto bilinearForm = make_BilinearForm(testSearchSpaces, solutionSpaces,
141             make_tuple(
142                 make_IntegralTerm<0,0,IntegrationType::valueValue,
143                                     DomainOfIntegration::interior>(c),
144                 make_IntegralTerm<0,0,IntegrationType::gradValue,
145                                     DomainOfIntegration::interior>(-1., beta),
146                 make_IntegralTerm<0,1,IntegrationType::normalVector,
147                                     DomainOfIntegration::face>(1., beta)));
148     auto innerProduct = make_InnerProduct(testSearchSpaces,
149             make_tuple(
150                 make_IntegralTerm<0,0,IntegrationType::valueValue,
151                                     DomainOfIntegration::interior>(1.),
152                 make_IntegralTerm<0,0,IntegrationType::gradGrad,
153                                     DomainOfIntegration::interior>(1., beta)));
154 
155     typedef GeometryBuffer<GridView::template Codim<0>::Geometry> GeometryBuffer;
156     GeometryBuffer geometryBuffer;
157 
158     auto systemAssembler
159         = make_DPGSystemAssembler(bilinearForm, innerProduct, geometryBuffer);
```

The `systemAssembler` has a member variable `testspaceCoefficientMatrix_` which is of type `TestspaceCoefficientMatrixBuffered` or `TestspaceCoefficientMatrixUnbuffered`, depending on whether `make_DPGSystemAssembler` is called with or without a `GeometryBuffer`, for details on that, see Section 3.1.4.

Once `systemAssembler` has been defined, a call to the method `assembleSystem(stiffnessMatrix, rhsVector, rhsFunctions)` assembles the matrix A and the right-hand side vector F. They are stored in the variables `stiffnessMatrix` (of type `BCRSMatrix<FieldMatrix<double,1,1>>`) and `rhsVector` (of type `BlockVector<FieldVector<double,1>>`). The input parameter `rhsFunctions` is of type `LinearForm` and represents the function f from the right hand side of the PDE. Internally, the class `DPGSystemAssembler` iterates over all mesh cells K and delegates the work of computing local contributions A_K to the system matrix A to `testspaceCoefficientMatrix_.systemMatrix()`. Similarly the local right-hand side vectors are computed by taking the product of the precomputed `testspaceCoefficientMatrix_.coefficientMatrix()` with `LinearForm::getLocalVector()`. For constructing A and F out of the local matrices A_K and the local right-hand side vectors, we make use of the mapping between local and global degrees of freedom given by the index sets from Dune-Functions.

`DPGSystemAssembler` is also responsible for applying boundary conditions to the system. So far, only Dirichlet boundary conditions are implemented. To this end, first the degrees of freedom affected by the boundary condition are marked. Then the boundary values are set to the corresponding nodes with the method `applyDirichletBoundary`. For instance, if we are considering the transport problem of Section 2.3, we need to set to zero the degrees of freedom of w that are in Γ_-. For this, we mark the relevant nodes with the method `getInflowBoundaryMask` and store the information in a vector `dirichletNodesInflow`. Then we call `applyDirichletBoundary` as we outline in the following listing. The sequence of instructions is given in the code below. Note that the trial space associated to w is required. Since, in our ordering, w is our second unknown, we get its associated trial space with the command `std::get<1>(solutionSpaces)` (since `std::tuple` starts counting from 0).

src/plot_solution.cc

```
165     typedef BlockVector<FieldVector<double,1> > VectorType;
166     typedef BCRSMatrix<FieldMatrix<double,1,1> > MatrixType;
167 
168     VectorType rhsVector;
169     MatrixType stiffnessMatrix;
```

```
170
171 auto rhsFunctions
172   = make_DPGLinearForm(testSearchSpaces,
173         std::make_tuple(make_LinearIntegralTerm<0,
174                              LinearIntegrationType::valueFunction,
175                              DomainOfIntegration::interior>(f(beta))));
176 systemAssembler.assembleSystem(stiffnessMatrix, rhsVector, rhsFunctions);
177
178 // Determine Dirichlet dofs for w (inflow boundary)
179 {
180   std::vector<bool> dirichletNodesInflow;
181   BoundaryTools boundaryTools = BoundaryTools();
182   boundaryTools.getInflowBoundaryMask(std::get<1>(solutionSpaces),
183                                       dirichletNodesInflow,
184                                       beta);
185   systemAssembler.applyDirichletBoundary<1>
186       (stiffnessMatrix,
187        rhsVector,
188        dirichletNodesInflow,
189        0.);
190 }
```

Finally, in certain types of problems, some degrees of freedom might be ill-posed. For example, in the transport case, the degrees of freedom corresponding to trial functions on faces aligned with the flow direction will be weighted with 0 coefficients in the matrix or interior degrees of freedom of the lifting w of the trace variable may be undefined. To address these issues, `DPGSystemAssembler` provides several methods like `defineCharacteristicFaces` to interpolate undefined degrees of freedom on characteristic faces or `applyMinimization` to handle undefined degrees of freedom in the interior and optionally also on characteristic faces by minimizing a given norm.

Once A and F are obtained, the system $Ax = F$ (which is, from the theory, invertible) can be solved with the user's favorite direct or iterative scheme.

3.1.2 BilinearForm and InnerProduct As it follows from (10), the bilinear form $b(\cdot,\cdot)$ can be decomposed into local bilinear forms $b_K(\cdot,\cdot)$. The `BilinearForm` class describes b_K and provides access to the corresponding local matrices A_K defined in (20) which are then used by the `DPGSystemAssembler` to assemble the global matrix A.

In our case, we view a bilinear form b_K as a sum of what we will call elementary integral terms. By this we mean integrals over K (or ∂K) which are a product of a test search function $v \in \mathbb{V}_h$ (or its derivatives) and a trial function $u \in \mathbb{U}_H$ (or its derivatives). Additionally, the product might also involve some given function c. The current release only supports constant functions c and the non-constant case will be delivered in a future release. For instance, in our transport equation (cf. (19)),

$$b_K(u,v) = b_K((\varphi,w),v) := \underbrace{\int_K c v\varphi}_{Int_0} - \underbrace{\int_K \beta\cdot\nabla v\varphi}_{Int_1} + \underbrace{\sum_{K_h\in\Omega_h, K_h\subset K}\int_{\partial K_h} vw\beta\cdot n}_{Int_2}, \tag{24}$$

where we have omitted the tilde to ease notation here. Therefore the matrix $A_K = \sum_{i\in I} A_K^i$ can be computed as a sum of the matrices A_K^i corresponding to the different elementary integrals Int_i, $i \in I$. Any of the elementary integrals can be expressed via the class `IntegralTerm` that we describe in Section 3.1.3.

To create an object `bilinearForm` of the class `BilinearForm`, we call `make_BilinearForm` as follows.

C++ code

```
1 auto bilinearForm = make_BilinearForm (testSearchSpaces, solutionSpaces, terms);
```

The variables testSearchSpaces and solutionSpaces are the ones introduced in Section 3.1.1 to represent $\mathbb{V}_h$ and $\mathbb{U}_H$. The object terms is a tuple of objects of the class IntegralTerm. Once that the object bilinearForm exists, a call to the method getLocalMatrix computes A_K by iterating over all elementary integral terms and summing up their contributions A^i_K.

Let us now briefly discuss the class InnerProduct. Its aim is to allow the computation of the inner products associated to the Hilbert spaces $\mathbb{U}$ and $\mathbb{V}$. For this, we take advantage of the fact that an inner product can be seen as a symmetric bilinear form $b(u,v)$ where u and v are both functions from some space. Hence, we can reuse the structure of BilinearForm for summing over elementary integral terms to define the class InnerProduct. The construction of an InnerProduct is thus done with

C++ code

```
auto innerProduct = make_InnerProduct (testSpaces, terms);
```

3.1.3 IntegralTerm

An IntegralTerm represents an elementary integral over the interior of a cell K, over its faces ∂K or even over faces of a partition of K. It expresses a product between a term related to a test function v and a term related to a trial function u. Examples are Int_0, Int_1 and Int_2 from (24).

The IntegralTerm is parametrized by two size_t that give the indices of the test and trial spaces that we want to integrate over. Additionally we specify the type of evaluations used in the integral with a template parameter of type

C++ code

```
enum class IntegrationType {
    valueValue,
    gradValue,
    valueGrad,
    gradGrad,
    normalVector,
    normalSign
};
```

and the domain of integration with a template parameter of type

C++ code

```
enum class DomainOfIntegration {
    interior,
    face
};
```

If integrationType is of type IntegrationType::valueValue or IntegrationType::normalSign, the function make_IntegralTerm has to be called as follows:

C++ code

```
auto integralTerm
   = make_IntegralTerm<lhsSpaceIndex, rhsSpaceIndex,
                       integrationType, domainOfIntegration>(c);
```

where c is a scalar coefficient in front of the test space product and is of arithmetic type, e. g. double. The template parameter domainOfIntegration is one of the types from DomainOfIntegration and the parameters lhsSpaceIndex and rhsSpaceIndex refer, in this particular order, to the indices of test and trial space in their respective tuples of test and trial spaces. Note that the objects of the class IntegralTerm are not given the spaces themselves but only some indices referring to them. This is because the spaces are managed by the class BilinearForm (or InnerProduct) owning the IntegralTerm.

For other `integrationTypes`, we also need to specify the flow direction `beta` by calling

C++ code

```
1  auto integralTerm
2     = make_IntegralTerm<lhsSpaceIndex, rhsSpaceIndex,
3                         integrationType, domainOfIntegration>(c, beta);
```

where `c` is again of arithmetic type and `beta` is of vector type, e. g. `FieldVector<double, dim>`.

The `IntegralTerm` Int_1 from example (24) can be created with

C++ code

```
1  auto integralTerm
2     = make_IntegralTerm<0, 0, IntegrationType::gradValue,
3                         DomainOfIntegration::interior>(-1., beta);
```

where the two zeroes are, in this particular order, the indices of test and trial space in their respective tuples of test and trial spaces.

The class `IntegralTerm` provides a method `getLocalMatrix` that computes its contribution A_K^i to the local matrix A_K and that is called by the `getLocalMatrix` method of `BilinearForm` or `InnerProduct`. Since one would normally want to write a program which solves a fixed problem, we use Boost Fusion[2] to do as much work at compile time as possible. Since the `IntegrationType` and `DomainOfIntegration` of each `IntegralTerm` are compile time constants, an optimizing C++ compiler should be able to optimize away some unused code branches. Defining the bilinear form and inner product at compile time also gives us the opportunity to check for errors in the problem formulation at compile time. For now this is not done, but we would like to include such checks in the future to improve the usability of our library.

3.1.4 TestspaceCoefficientMatrix (buffered and unbuffered)

For any element K, the computation of the matrices C_K and A_K defined in (22) and (23) is handled either by the class `TestspaceCoefficientMatrixBuffered` or by `TestspaceCoefficientMatrixUnbuffered`. Two classes have been developed to minimize computations when the PDE coefficients are constant. In this case C_K and A_K depend only on the geometry of the element K. As a result, the value of C_K and A_K will be constant for all elements K having, up to a translation, the same map to the reference element. Thus, C_K and A_K can be computed only once for all cells sharing this mapping property. The class `TestspaceCoefficientMatrixBuffered` makes use of the above and reduces computational costs in grids where many cells have the same mapping property. The class `TestspaceCoefficientMatrixUnbuffered` handles the general case (grids with many different cell mapping types and, in future releases, non-constant PDE coefficients).

Structurally speaking, both classes have the bilinear form $b(\cdot,\cdot)$ and the inner product $\langle\cdot,\cdot\rangle_{\mathbb{V}}$ as template parameters and offer a method `bind(const Entity& e)` in which they use the methods `BilinearForm::getLocalMatrix()` and `InnerProduct::getLocalMatrix()` to set up B_K and G_K as defined in (21). They compute $C_K = G_K^{-1} B_K$ via the Cholesky algorithm and $A_K = B_K^T C_K$. The computed matrices can be accessed via the methods `coefficientMatrix()` and `systemMatrix()`, respectively. The constructor of `TestspaceCoefficientMatrixUnbuffered` gets only the bilinear form and the inner product. In addition to this, `TestspaceCoefficientMatrixBuffered` needs a `GeometryBuffer` containing a map to save the geometry of the elements K and the corresponding matrices C_K and A_K. When the `TestspaceCoefficientMatrixBuffered` is bound to a new element K, then first it is checked whether its map to the reference element is already in the buffer. If so, the saved matrices C_K and A_K are used. Otherwise, they are computed and added to the map.

[2] Fusion is a meta programming library and part of the C++ library collection Boost: http://www.boost.org/

3.2 A posteriori error estimators

To compute the residual

$$\|f - \mathcal{B}u_H\|_{\mathbb{V}'} = \sup_{v \in \mathbb{V}} \frac{\|f(v) - b(u_H, v)\|_{\mathbb{V}}}{\|v\|_{\mathbb{V}}},$$

we exploit once again the product structure of $\mathbb{V}$ and use the fact that

$$\|f - \mathcal{B}u_H\|^2_{\mathbb{V}'} = \sum_{K \in \Omega_H} \|r_K(u_H, f)\|^2_{\mathbb{V}'_K} = \sum_{K \in \Omega_H} \|R_K(u_H, f)\|^2_{\mathbb{V}_K}$$

where r_K is the cell-wise residual. R_K is the Riesz-lift of r_K in $\mathbb{V}_K$ so it is the solution of

$$\langle R_K(u_H, f), v\rangle_{\mathbb{V}_K} = b(u_H, v) - f(v), \quad \forall v \in \mathbb{V}_K. \tag{25}$$

Since (25) is an infinite-dimensional problem, we project the Riesz-lift R_K to a finite-dimensional subspace $\overline{\mathbb{V}}_K$ of $\mathbb{V}_K$, obtaining an approximation $\overline{R}_K$. This in turn gives the a posteriori error estimator

$$\|\overline{R}(u_H, f)\|_{\mathbb{V}} := \left(\sum_{K \in \Omega_H} \|\overline{R}_K(u_H, f)\|^2_{\mathbb{V}_K} \right)^{1/2}. \tag{26}$$

An appropriate choice of the a posteriori search space $\overline{\mathbb{V}}_K$ depends on the problem and is crucial to make $\|\overline{R}_K\|_{\mathbb{V}_K}$ good error indicators.

In Dune-DPG, the computation of the a posteriori estimator (26) is handled by the class `ErrorTools`. The following lines of code compute (26) for the solution `u_H` of a problem with bilinear form `bilinearForm`, inner product `innerProduct` and right hand side `rhsVector`.

C++ code

```
1 ErrorTools errorTools = ErrorTools();
2 double aposterioriErr =
      errorTools.aPosterioriError(bilinearForm,innerProduct,u_H,rhsVector);
```

The object `bilinearForm` is of the type `BilinearForm` described above. It has to be created with an object `testSpace` associated to the a posteriori search space $\overline{\mathbb{V}}_H$. The same applies for `innerProduct`, which is of type `InnerProduct`. Also the vector `rhsVector` has to be set up using the a posteriori search space.

Furthermore, `ErrorTools` contains a method `DoerflerMarking` to use the approximate cell-wise residuals $\|\overline{R}_K\|_{\mathbb{V}_K}$ for adaptive refinement.

4 Numerical Example: Implementation of Pure Transport in Dune-DPG

As a simple numerical example, we solve the transport problem (13) with

$$\begin{aligned} c &= 0, \\ \beta &= (\beta_1, \beta_2) = (\cos(\pi/8), \sin(\pi/8)), \\ f &= 1. \end{aligned}$$

As Figure 1d shows, the exact solution

$$\varphi(x) = \begin{cases} \sqrt{\beta_2^2/\beta_1^2 + 1} \cdot x_1, & \text{if } \beta_1 \cdot x_2 - \beta_2 \cdot x_1 > 0 \\ \sqrt{\beta_1^2/\beta_2^2 + 1} \cdot x_2, & \text{else} \end{cases}$$

describes a linear ramp starting at 0 in each point of the inflow boundary Γ_- and increasing with slope 1 along the flow direction β. There is a kink in the solution starting in the lower left corner of Ω and propagating along β.

For the numerical solution, we let Ω_H be a partition of Ω into uniformly shape regular triangles generated by partitioning $\Omega = [0,1]^2$ into $n \times n$ uniform quadrangles that are then each divided from the lower left to the upper right corner into two triangles. Ω_h is a refinement of Ω_H to some level $\ell \in \mathbb{N}_0$ such that $h = 2^{-\ell}H$, i. e. each triangle in Ω_H gets subdivided into 4^ℓ congruent subtriangles.

With $\mathbb{U}_H$ and $\mathbb{V}_h$ defined as in (16) and (17) with $m = 2$, we compute $u_H = (\varphi_H, w_H) \in \mathbb{U}_H$ by solving the ultra-weak variational formulation (18). We investigate convergence in H of the error $\|\varphi - \varphi_H\|_{L_2(\Omega)}$. We also evaluate the a posteriori estimator $\|\overline{R}(u_H, f)\|_{\mathbb{V}}$ when the components $\overline{R}_K(u_H, f)$ are computed with a subspace $\overline{\mathbb{V}}_K$ of polynomials of degree 5, $\forall K \in \Omega_H$.

Regarding the error $\|\varphi - \varphi_H\|_{L_2(\Omega)}$, as Figure 1a shows, we observe linear convergence as H decreases. This is to be expected since the polynomial degree to compute φ_H is 1. The figure also shows that the refinement level ℓ of the test-search space $\mathbb{V}_h$ has essentially no impact on the behavior of the error.

Regarding the behavior of the a posteriori estimator $\|\overline{R}(u_H, f)\|_{\mathbb{V}}$, it is possible to see in Figures 1b and 1c that the quality of $\|\overline{R}(u_H, f)\|_{\mathbb{V}}$ slightly degrades as H decreases in the sense that, as H decreases, $\|\overline{R}(u_H, f)\|_{\mathbb{V}}$ represents the error $\|\varphi - \varphi_H\|_{L_2(\Omega)}$ less and less faithfully. An element that might be playing a role is that $\|\overline{R}_K(u_H, f)\|_{\mathbb{V}}$ is not exactly an estimation of $\|\varphi - \varphi_H\|_{L_2(\Omega)}$, but of the error including also w_H, namely $\|u - u_H\|_{\mathbb{U}} = \|(\varphi, w) - (\varphi_H, w_H)\|_{\mathbb{U}}$.

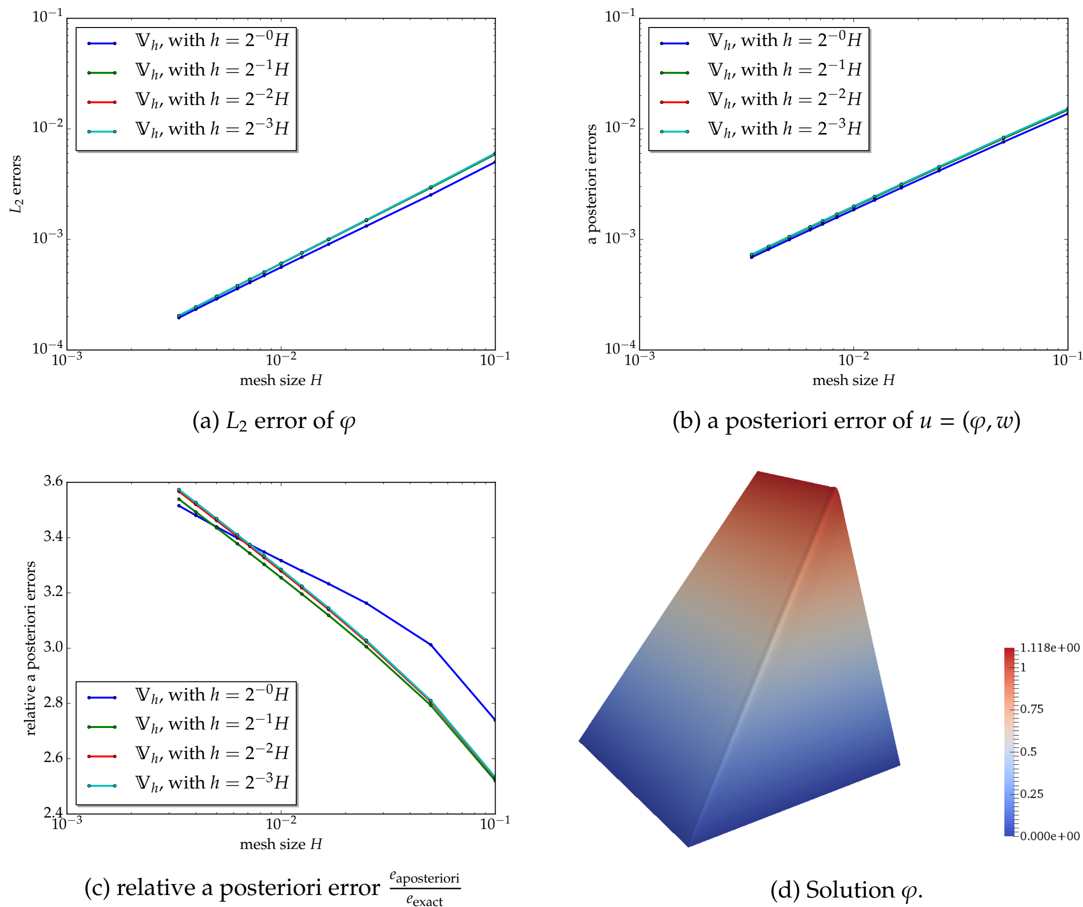
The source code used to compute the values from the convergence plots in Figure 1a–c can be found in `src/convergence_test.cc` while the code used to generate Figure 1d can be found in `src/plot_solution.cc`. In this program, the user can easily experiment with the transport equation by changing the values of β and c (see the `README` file).

In Figure 2 we compare the runtime for assembling the system with and without buffering of the test space coefficient matrices from Section 3.1.4 on a 4 core Intel Core i7-3770 CPU with 3.40GHz and 15GB of RAM. These performance measurements have been done for the same problem as before and the corresponding code resides in `src/profile_testspacecoefficientmatrix.cc`. The plots show the clear advantage of using the buffering whenever the grid is uniform and we use constant coefficients.

5 Conclusion And Future Work

In Section 2, we gave a short overview of the DPG method. We then introduced our Dune-DPG library in Section 3, documenting the internal structure and showing how to use it to solve a given PDE. Finally, we showed some numerical convergence results computed for a problem with well-known solution. This allowed us to compare our a posteriori estimators to the real L_2 error of our numerical solution. As a next step, we want to fully implement non-constant coefficients in order to support a wider range of PDEs.

Furthermore, we want to improve our handling of vector valued problems with one notable example being first order formulations of convection–diffusion problems. With our current `std::tuple` of spaces structure used throughout the code, we have to implement vector valued spaces by adding the same scalar valued space several times. With the Dune-TypeTree library from Müthing [2015] we can handle vector valued spaces much more easily, as has already been shown in Dune-Functions. This will result in major changes in our code, but will probably allow us to replace our dependency on Boost Fusion with more modern C++11 constructs. In the long run, we hope that this would give us increased maintainability and decreased compile times in addition to the improvements in the usability of vector valued problems. Finally, we want to explore parallelization to increase the performance. This is aligned with our long-term goal of



(a) L_2 error of φ

(b) a posteriori error of $u = (\varphi, w)$

(c) relative a posteriori error $\frac{e_{\text{aposteriori}}}{e_{\text{exact}}}$

(d) Solution φ.

Figure 1: L_2 error and a posteriori error estimator of numerical solutions

making DUNE-DPG a flexible building block for constructing DPG solvers for a large range of different problem types.

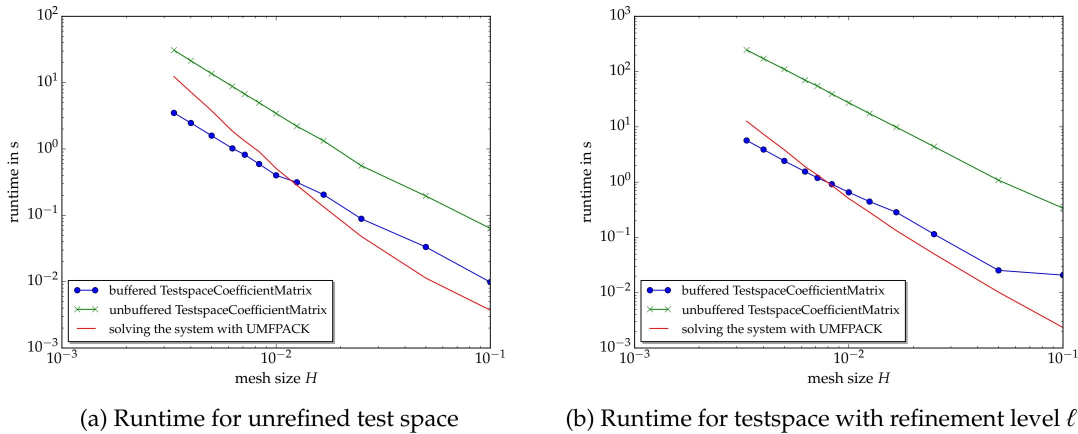
Acknowledgments

We thank O. Sander for his introduction to the DUNE library and his guidance in understanding it. We also thank W. Dahmen for introducing us to the topic of DPG and our anonymous reviewers for their constructive comments which helped to significantly improve the paper and the efficiency of the computation of test functions in our code. Finally, O. Mula is indebted to the AICES institute of RWTH Aachen for hosting her as a postdoc during 2014–2015 which is the period in which large parts of DUNE-DPG were developed.

References

P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuß, H. Rentz-Reichert, and C. Wieners. UG – a flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science*, 1(1):27–40, 1997. doi: 10.1007/s007910050003.

P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE. *Computing*, 82(2–3):121–138, 2008. doi: 10.1007/s00607-008-0004-9.



(a) Runtime for unrefined test space

(b) Runtime for testspace with refinement level $\ell = 2$

Figure 2: Runtime for assembling the system with buffered and unbuffered testspace coefficient matrices.

M. Blatt and P. Bastian. The iterative solver template library. In *International Workshop on Applied Parallel Computing*, pages 666–675. Springer, 2006. doi: 10.1007/978-3-540-75755-9_82.

M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, and O. Sander. The Distributed and Unified Numerics Environment, version 2.4. *Archive of Numerical Software*, 4(100):13–29, 2016. doi: 10.11588/ans.2016.100.26526.

D. Broersen and R. Stevenson. A robust Petrov–Galerkin discretisation of convection–diffusion equations. *Computers & Mathematics with Applications*, 68(11):1605–1618, 2014. doi: 10.1016/j.camwa.2014.06.019.

D. Broersen and R. P. Stevenson. A Petrov–Galerkin discretization with optimal test space of a mild-weak formulation of convection-diffusion equations in mixed form. *IMA Journal of Numerical Analysis*, 35(1):39–73, 2015. doi: 10.1093/imanum/dru003.

D. Broersen, W. Dahmen, and R. P. Stevenson. On the stability of DPG formulations of transport equations. IGPM Preprint 433, IGPM, RWTH Aachen, Oct. 2015. URL https://www.igpm.rwth-aachen.de/forschung/preprints/433.

C. Carstensen, L. Demkowicz, and J. Gopalakrishnan. A posteriori error control for DPG methods. *SIAM Journal on Numerical Analysis*, 52(3):1335–1353, 5 June 2014. doi: 10.1137/130924913.

C. Carstensen, L. Demkowicz, and J. Gopalakrishnan. Breaking spaces and forms for the DPG method and applications including Maxwell equations. *Computers & Mathematics with Applications*, 72(3):494–522, 2016. doi: 10.1016/j.camwa.2016.05.004.

W. Dahmen, C. Huang, C. Schwab, and G. Welper. Adaptive Petrov–Galerkin methods for first order transport equations. *SIAM Journal on Numerical Analysis*, 50(5):2420–2445, 2012. doi: 10.1137/110823158.

W. Dahmen, C. Plesken, and G. Welper. Double greedy algorithms: Reduced basis methods for transport dominated problems. *ESAIM: Mathematical Modelling and Numerical Analysis*, 48(3):623–663, 2014. doi: 10.1051/m2an/2013103. URL http://www.igpm.rwth-aachen.de/forschung/preprints/357.

L. Demkowicz and J. Gopalakrishnan. A class of discontinuous Petrov–Galerkin methods. Part II: Optimal test functions. *Numerical Methods for Partial Differential Equations*, 27(1):70–105, Jan. 2011. doi: 10.1002/num.20640.

L. Demkowicz and J. Gopalakrishnan. Discontinuous Petrov–Galerkin (DPG) method. ICES Report 15-20, ICES, UT Austin, Oct. 2015. URL https://www.ices.utexas.edu/media/reports/2015/1520.pdf.

L. Demkowicz, J. Gopalakrishnan, I. Muga, and J. Zitelli. Wavenumber explicit analysis of a DPG method for the multidimensional Helmholtz equation. *Computer Methods in Applied Mechanics and Engineering*, 213–216:126–138, Mar. 2012. doi: 10.1016/j.cma.2011.11.024.

C. Engwer, C. Gräser, S. Müthing, and O. Sander. The interface for functions in the dune-functions module. Dec. 2015. URL http://arxiv.org/abs/1512.06136.

J. Gopalakrishnan and W. Qiu. An analysis of the practical DPG method. *Mathematics of Computation*, 83(286):537–552, 2014. doi: 10.1090/S0025-5718-2013-02721-4. URL http://arxiv.org/abs/1107.4293.

S. Müthing. *A Flexible Framework for Multi Physics and Multi Domain PDE Simulations*. PhD thesis, Universität Stuttgart, Feb. 2015.

A. H. Niemi, N. O. Collier, and V. M. Calo. Automatically stable discontinuous Petrov–Galerkin methods for stationary transport problems: Quasi-optimal test space norm. *Computers & Mathematics with Applications*, 66(10):2096–2113, 2013. doi: 10.1016/j.camwa.2013.07.016.

N. V. Roberts. Camellia: A software framework for discontinuous Petrov–Galerkin methods. *Computers & Mathematics with Applications*, 68(11):1581–1604, 2014. doi: 10.1016/j.camwa.2014.08.010.

N. V. Roberts, T. Bui-Thanh, and L. Demkowicz. The DPG method for the Stokes problem. *Computers & Mathematics with Applications*, 67(4):966–995, 2014. doi: 10.1016/j.camwa.2013.12.015.