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

Felix Gruber, Angela Klewinghaus, Olga Mula

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, CEREMADE, 75775 Paris, France

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. This guarantees that residuals can be used to tightly bound errors from below and above and is crucial for a posteriori error control and the development of adaptive strategies. In this framework, the so-called Discontinuous Petrov–Galerkin (DPG) concept can be viewed as a systematic strategy of contriving variational formulations which possess these desirable stability properties, see e. g. Broersen et al. [2015]. In this paper, we present a C++ library, DUNE-DPG, which serves to implement and solve such variational formulations. The library is built upon the multi-purpose finite element package DUNE (see Blatt et al. [2016]). One of the main features of DUNE-DPG is its flexibility which is achieved by a highly modular structure. The library can solve in practice some important classes of PDEs (whose range goes beyond classical second order elliptic problems and includes e. g. transport dominated problems). As a result, DUNE-DPG can also be used to address other problems like optimal control with the DPG approach.

1 Introduction

General context and motivations: 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 $\mathcal{B} : \mathbb{U} \to \mathbb{V}$ be a boundedly invertible linear operator and let $b : \mathbb{U}\times\mathbb{V} \to \mathbb{R}$ 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 Galekin 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, much less is known when $\mathbb{U} \neq \mathbb{V}$ is required to obtain a well-posed and well-conditioned problem, like for transport-dominated PDEs. In the latter case, the main ideas are that

- $\gamma = 1$ by choosing a problem-dependent norm for $\mathbb{V}$ and
- for a given finite-dimensional trial space $\mathbb{U}_H$, there exists a corresponding optimal test space $\mathbb{V}^{opt}(\mathbb{U}_H)$ such that the discrete inf-sup condition

$$\inf_{w_H \in \mathbb{U}_H} \sup_{v \in \mathbb{V}^{\mathrm{opt}}(\mathbb{U}_H)} \frac{b(w_H, v)}{\|w_H\|_{\mathbb{U}}\|v\|_{\mathbb{V}}} \geq \gamma \tag{4}$$

  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]). In this method, an approximation of the exact optimal test functions is realized in the context of a discontinuous Petrov–Galerkin formulation. This yields a so-called near-optimal test space which is the one that is eventually used in the solution of the discrete problem. The tightness of a posteriori error estimators has been theoretically justified only for second order elliptic problems. However, numerical evidence illustrates their good performance also for a much broader variety of problems. Without being exhaustive, we can find works on transport equations [Broersen et al., 2015], convection–diffusion [Broersen and Stevenson, 2015], elasticity and Stokes problems [Carstensen et al., 2014], Maxwell equations [Carstensen et al., 2015] and the Helmholtz equation [Demkowicz et al., 2012].

The presented finite element library Dune-DPG serves to implement and solve such DPG formulations.

Contributions and layout of the paper: In this paper, we explain the construction of the Dune-DPG library which is capable of solving various types of PDEs with a DPG variational formulation. Since the appropriate characteristics of the formulation depend on the problem, the user is given the freedom to choose the spaces $\mathbb{U}$, $\mathbb{V}$, their finite-dimensional counterparts and the geometry and mesh refinement. We would like to note that, in fact, Dune-DPG is not the first software tool for solving PDEs with the DPG method. The *Camellia* package (see Roberts [2014]) is another library whose purpose and construction is similar to Dune-DPG. In Dune-DPG, we incorporate, at the software level, the latest theoretical results on DPG for transport equations, given by

Broersen et al. [2015], that allow to appropriately address families of transport-based PDEs. They essentially require the use of subgrids of fixed depth for the finite-dimensional test space. For this reason, in Dune-DPG, a strong emphasis has been put on providing as much freedom as possible in the selection of the finite-dimensional test space. This makes Dune an appropriate choice of foundation for our library since its modular structure gives low-level access to those parts for which fine control is required while still providing high-level functionality for the rest of the code.

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.

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 $\mathbb{V}^{\mathrm{opt}}(\mathbb{U}_H)$ of the same dimension. It is called optimal because the finite-dimensional version of problem (2),

$$\begin{aligned}&\text{Find } u_H \in \mathbb{U}_H \text{ s. t.}\\ &b(u_H, v) = f(v), \quad \forall v \in \mathbb{V}^{\mathrm{opt}}(\mathbb{U}_H),\end{aligned} \tag{5}$$

is well posed and

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

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,

$$i \in \{1, \ldots, \mathcal{N}\}, \quad \langle v^i, v\rangle_{\mathbb{V}} = b(u_H^i, v), \quad \forall v \in \mathbb{V} \tag{6}$$

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 (6) 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 $i \in \{1, \dots, \mathcal{N}\}$,

$$\langle \bar{v}^i, v \rangle_{\mathbb{V}} = b(u_H^i, v), \quad \forall v \in \mathbb{V}_h. \tag{7}$$

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

$$\gamma_H := \inf_{u_H \in \mathbb{U}_H} \sup_{v_h \in \mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)} \frac{b(u_H, v_h)}{\|u_H\|_{\mathbb{U}_H} \|v_h\|_{\mathbb{V}_h}} \tag{8}$$

is bounded away from 0 uniformly in H. For this reason, $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ 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 $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ is the one that is computed in practice in the Dune-DPG library. The finite-dimensional variational formulation that is eventually solved reads

$$\begin{aligned} &\text{Find } u_H \in \mathbb{U}_H \text{ s. t.} \\ &b(u_H, v_h) = f(v_h), \quad \forall v_h \in \mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h). \end{aligned} \tag{9}$$

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 (7) to derive the near-optimal basis functions of $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ 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 (9) 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,

$$\mathbb{V} := \prod_{K \in \Omega_H} \mathbb{V}_K, \tag{10}$$

where $\text{supp}(v) \subset K$ 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 choosen 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 $1 \le i \le \mathcal{N}$, the near-optimal test function $\bar{v}^i$ can be written as

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

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,

$$b(u, v) = \sum_{K \in \Omega_H} b_K(u, v), \quad \forall v \in \prod_{K \in \Omega_H} \mathbb{V}_K. \tag{11}$$

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

$$\langle \bar{v}_K^i, v \rangle_{\mathbb{V}_K} = b_K\left(u_H^i, v\right), \ \forall v \in \mathbb{V}_{h,K}, \tag{12}$$

where $\{u_H^i\}_{i=1}^N$ 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_H^i 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 $\bar{v}^i = \bar{v}_K^i \chi_K$ or $\bar{v}^i = \sum_{K' \in \text{neigh(K)}} \bar{v}_{K'}^i \chi_{K'}$. Hence, if the basis functions u_H^i 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

$$\Gamma_- := \{x \in \partial\Omega \mid \beta \cdot n(x) < 0\} \subset \partial\Omega \tag{13}$$

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

$$\begin{aligned} \beta \cdot \nabla\varphi + c\varphi &= f, \quad \text{in } \Omega, \\ \varphi &= 0, \quad \text{on } \Gamma_-. \end{aligned} \tag{14}$$

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 ∇_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 $\|v\|_{H(\beta,\Omega_H)}^2 = \|v\|_{L_2(\Omega)}^2 + \|\nabla_H \cdot v\|_{L_2(\Omega)}^2$. 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

$$\begin{aligned} &\text{For } \mathbb{U} := L^2(\Omega) \times H_{0,\Gamma_-}(\beta, \partial\Omega_H) \text{ and } \mathbb{V} := H(\beta, \Omega_H), \\ &\text{given } f \in H(\beta, \Omega_H)', \text{ find } u := (\varphi, \theta) \in \mathbb{U} \text{ such that} \\ &b(u, v) = f(v), \quad \forall v \in \mathbb{V}. \end{aligned} \tag{15}$$

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

$$b(u, v) = b((\varphi, \theta), v) = \int_\Omega (-\beta \cdot \nabla v\varphi + cv\varphi) \ dx + \int_{\partial\Omega_H} [\![v\beta]\!]\theta \ ds. \tag{16}$$

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 take for some $m \in \mathbb{N}$,

$$\mathbb{U}_H := \Big(\prod_{K\in\Omega_H} \mathbb{P}_{m-1}(K) \Big) \times \Big(H_{0,\Gamma_-}(\beta;\Omega) \cap \prod_{K\in\Omega_H} \mathbb{P}_m(K) \Big)\Big|_{\partial\Omega_h}, \tag{17}$$

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

$$\mathbb{V}_h := \prod_{K\in\Omega_h} \mathbb{P}_{m+1}(K). \tag{18}$$

As a result, the discrete version of (15) reads

$$\begin{aligned} &\text{Find } u_H := (\varphi_H, \theta_H) \in \mathbb{U}_H \text{ such that} \\ &\tilde{b}(u_H, v_h) = f(v_h), \quad \forall v_h \in \mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h). \end{aligned} \tag{19}$$

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

$$\tilde{b}(u,v) = b\left((\varphi,\theta),v\right) = \int_\Omega (-\beta\cdot\nabla v\varphi + cv\varphi)\ \mathrm{d}x + \int_{\partial\Omega_h} [\![v\beta]\!]\theta\ \mathrm{d}s, \tag{20}$$

where the trace integral is over $\partial\Omega_h$ and not $\partial\Omega_H$.

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 present library has been built upon the finite element package Dune. It benefits from the new Dune-Functions module [Engwer et al., 2015] that has been critical for the construction of a uniform interface for test and trial spaces.

The user of Dune-DPG 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). Next, the near-optimal test space is determined automatically with the help of the bilinear form and the inner product via (12) (see Section 3.2). Finally, the `SystemAssembler` class handles the automatic assembly of the linear system $Ax = F$ associated to problem (9) including the right hand side f and boundary conditions (see Section 3.1.1). For assembling, the matrix A, we use (11) and define the local matrices A_K by

$$(A_K)_{i,j} = b_K(u^i_{H,K}, \tilde{v}^j_K) \tag{21}$$

where $\{u^i_{H,K}\}_{i=1}^{N_K}$ is a basis for the restriction of $\mathbb{U}_H$ to K and $\{\tilde{v}^j_K\}_{j=1}^{N_K}$ is a basis for the restriction of the near-optimal test spaces $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ to K. The `BilinearForm` class provides the local matrices A_K and the `SystemAssembler` class constructs the global matrix A out of the local matrices A_K. The classes that we have just mentioned are intertwined and depend on each other. Figure 1 gives an overview of their interactions. In addition to these classes, the class `ErrorTools` handles the computation of a posteriori estimators following the guidelines that are given in Section 3.3.

3.1 Assembling the discrete system for a given PDE

The following subsections describe the `SystemAssembler` class and all the classes used by it, except for the `OptimalTestBasis` class that will be explained in detail in Section 3.2.

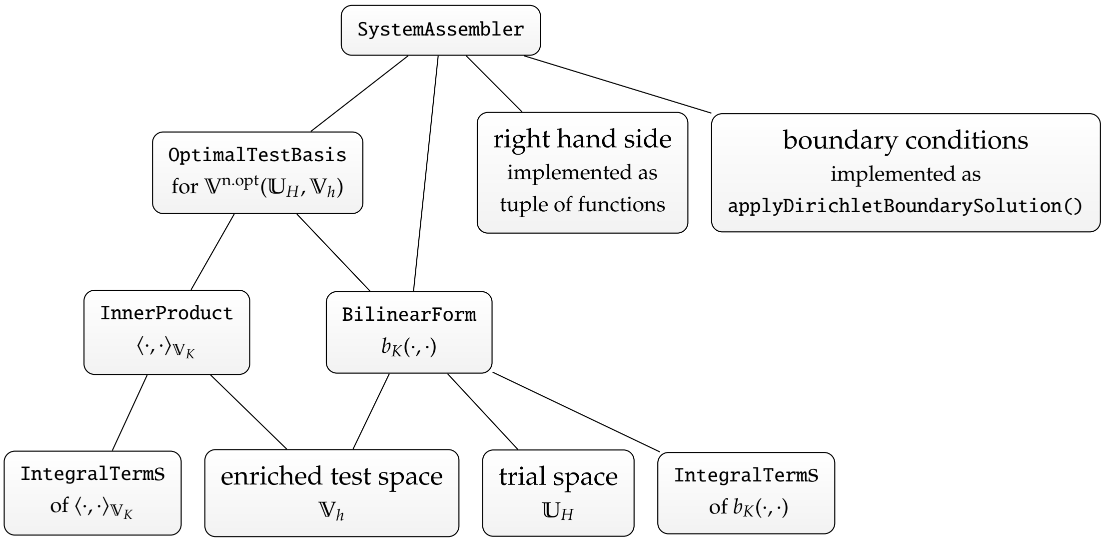


Figure 1: Overview of interactions between the main classes of Dune-DPG.

3.1.1 SystemAssembler The assembly of the discrete system $Ax = F$ derived from the variational problem is handled by the class `SystemAssembler`. We start by creating the appropriate object of the class (that we will name in our case `systemAssembler`) by calling the method `make_DPG_SystemAssembler`. As an input, it needs objects representing our trial space $\mathbb{U}_H$, our near-optimal test space $\mathbb{V}^{\text{n.opt}}(\mathbb{U}_H, \mathbb{V}_h)$ and the bilinear form b. 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 respectively given by a `std::tuple` composed of 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, θ) and $\mathbb{U}_H$ is a product of two spaces.

Once `systemAssembler` has been defined, a call to the method `assembleSystem(stiffnessMatrix, rhsVector, rhsFunction)` 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 `rhsFunction` is a `std::tuple` of `std::function<double(Dune::FieldVector<double,dim>)>` and represents the function f from the PDE. Internally, the class `SystemAssembler` iterates over all mesh cells K and delegates the work of computing local contributions A_K to the system matrix A to `BilinearForm::getLocalMatrix()`. Similarily the local right-hand side vectors are computed by a function `getVolumeTerm`. 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.

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. As a summary, the following lines of code give the main guidelines to solve the transport problem of Section 2.3. The commented lines starting with "[...]" mean that there is some code to be written in addition. We do not provide it for the sake of clarity and refer to our source code for an example of exact implementation (see `src/plot_solution.cc`).

C++ code

```cpp
// Definition of the trial spaces
DuneFEM1 spacePhi;
DuneFEM2 spaceTheta;
auto solutionSpaces = std::make_tuple(spacePhi, spaceTheta);
```

```
// [...] Definition of testSearchSpaces (tuple containing the test search space)
// [...] Computation the associated near optimal test space. The space is stored in
    the object nearOptTestSpaces (see Section 3.2)
// [...] Definition of the object bilinearForm (see Section 3.1.2)
// Creation of the object systemAssembler
auto systemAssembler
   = make_DPG_SystemAssembler(nearOptTestSpaces, solutionSpaces, bilinearForm);
// Assemble the system Ax=f
MatrixType stiffnessMatrix;
VectorType rhsVector;
//[...] Define rhsFunction as a suitable lambda expression
systemAssembler.assembleSystem(stiffnessMatrix, rhsVector, rhsFunction);
// [...] Compute ‘‘inverse(matrix)*rhsVector’’
```

SystemAssembler 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 applyDirichletBoundarySolution. For instance, if we are considering the transport problem of Section 2.3, we need to set the degrees of freedom of θ, that are in Γ_-, to 0. For this, we mark the relevant nodes with the method getInflowBoundaryMask and store the information in a vector dirichletNodesInflow. Then we call applyDirichletBoundarySolution as we outline in the following listing. Note that the trial space associated to θ is required. Since, in our ordering, θ is our second unknown, we get its associated trial space with the command std::get<1>(solutionSpaces) (since std::tuple starts counting from 0).

C++ code

```
std::vector<bool> dirichletNodesInflow;
BoundaryTools boundaryTools = BoundaryTools();
boundaryTools.getInflowBoundaryMask(std::get<1>(solutionSpaces),
                                    dirichletNodesInflow,
                                    beta);     //mark affected degrees of freedom
systemAssembler.applyDirichletBoundarySolution<1>
    (stiffnessMatrix,
     rhs,
     dirichletNodesInflow,
     0.);
```

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. To address this issue, SystemAssembler provides several methods, of which the simplest is defineCharacteristicFaces.

3.1.2 BilinearForm and InnerProduct

As it follows from (11), 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 (21) which are then used by the SystemAssembler 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 coefficient $c(x)$. For instance, in our transport equation (cf. (20)),

$$b_K(u,v) = b_K((\varphi,\theta),v) := \underbrace{\int_K c v\varphi}_{Int_0} - \underbrace{\int_K \beta \cdot \nabla v\varphi}_{Int_1} + \underbrace{\sum_{K_h \in \Omega_h, K_h \subset K} \int_{\partial K_h} v\theta\beta \cdot n}_{Int_2}, \tag{22}$$

where we have omitted the tilde to ease notation here. Therefore the matrix $A_K = \sum_{i \in I} A^i_K$ can be computed as a sum of the matrices A^i_K 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

```
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_K^i.

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 (22).

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

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

where `c` is again of arithmetic type and `beta` is of vector type, e. g. `FieldVector<double, dim>`. There has been some rudimentary work to support functions mapping coordinates to scalars or vectors for `c` and `beta`.

The `IntegralTerm` Int_1 from example (22) can be created with

C++ code

```cpp
auto integralTerm
   = make_IntegralTerm<0, 0, IntegrationType::gradValue,
                       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`. To prevent runtime switches over the `IntegrationType` and `DomainOfIntegration`, we made them template parameters of `IntegralTerm` and use Boost Fusion[1] to easily handle compile time abstractions.

3.2 Computing the optimal test space

As described in Section 2.2, for a given basis function u_H^i of the trial space $\mathbb{U}_H$, the corresponding near-optimal test function $\bar{v}^i = \sum_{K\in\Omega_H} \bar{v}_K^i \chi_K$ can be computed cell-wise. Indeed, one can find $\bar{v}_K^i$ by solving (12) for every $K \in \Omega_H$. As a consequence, we can decompose the computation of $\bar{v}^i$ into the following steps (we will omit the index i and call these functions u_H, $\bar{v}$ and $\bar{v}_K$ in the rest of this section):

- For a given cell $K \in \Omega_H$, let $\{z^j\}_{j=1}^M$ be a basis of $\mathbb{V}_{h,K}$, the test-search space on cell K. In this basis, we can express $\bar{v}_K = \sum_{j=1}^M c_K^j z^j$ and find the vector of coefficients $c_K = (c_K^j)_{j=1}^M$ as follows. Let us denote B_K the $\mathbb{R}^{M\times M}$ matrix with entries $(B_K)_{j,l} = \langle z^j, z^l\rangle_{\mathbb{V}_K}$ for $1 \le j,l \le M$ and $g_K \in \mathbb{R}^M$ the vector with entries $g_K^j = b_K(u_H, z^j)$, $1 \le j \le M$. Then c_K is the solution of the system
$$B_K c_K = g_K. \tag{23}$$
This task is done by the class `TestspaceCoefficientMatrix` for all basis functions u_H with $\operatorname{supp}(u_H) \cap K \neq \emptyset$ (see Section 3.2.1 for more details).

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

- Let K_{ref} be the reference cell and $\{z^j_{ref}\}_{j=1}^M$ the basis functions of $\mathbb{V}_{h,K}$ on K_{ref}. Having computed c_K, we define the corresponding local basis function of the near-optimal test space $\bar{v}_{K,ref} := \sum_{j=1}^M c^j_K z^j_{ref}$. This task is done by the class `OptimalTestLocalFiniteElement` (see Section 3.2.2).

- Having computed $\bar{v}_{K,ref}$ and using the degree of freedom-handling from the global basis of the trial space $\mathbb{U}_H$, we build $\bar{v}_{ref} = \sum_{K\in\Omega_H} \bar{v}_{K,ref}$. This task is done by the class `OptimalTestBasis` (see Section 3.2.3). Note that the remaining mapping from $\bar{v}_{ref}$ to $\bar{v}$ has to be performed for any global basis. It is done with the help of geometry-information while assembling the system matrix A and right hand side F.

3.2.1 TestspaceCoefficientMatrix

The computation of the coefficients c_K is performed by the class `TestspaceCoefficientMatrix` which has the bilinear form $b(\cdot,\cdot)$ and the inner product $\langle\cdot,\cdot\rangle_{\mathbb{V}}$ as template parameters. It has a method `bind(const Entity& e)` in which it sets up and solves equation (23) for all local basis functions u_K of the trial space. Since the matrix B_K is symmetric positive definite, the solution is determined via the Cholesky algorithm. The computed coefficients c_K are saved in a matrix which can be accessed by the method `coefficientMatrix()`.

Furthermore, `TestspaceCoefficientMatrix` offers the possibility to save and reuse already computed coefficients. In case of constant parameters, equation (23) only depends on the geometry of the cell K, so in case of a uniform grid, the coefficients are the same for all cells and thus need not be recomputed for every cell. To this end, in the current version, constant parameters are assumed and the geometry of the last cell is saved and compared to the geometry of the current cell. If they coincide, computation is skipped and the old coefficients are used. This is also helpful in cases with more than one or vector-valued test variables (see Section 3.2.3).

3.2.2 OptimalTestLocalFiniteElement

The class `OptimalTestLocalFiniteElement` provides a local basis $\{\bar{v}^i_{K,ref}\}_{i=1}^N$ consisting of linear combinations $\bar{v}^i_{K,ref} = \sum_{j=1}^M (c^i_K)^j z^j_{ref}$ of a given local basis $\{z^j_{ref}\}_{j=1}^M$. Its constructor is called as follows

C++ code

```
OptimalTestLocalFiniteElement< D, R, d, TestSearchSpace>
                    (coefficientMatrix, //matrix containing the coefficients c_K^i
                     testSearchSpace,   //given local basis {z_ref^j}
                     offset=0)          //optional, default-value: 0
```

The template parameters are the type `D` used for domain coordinates, the type `R` used for function values, an integer `d` specifying the dimension of the reference element and the type of the given local basis `TestSearchSpace`. The optional parameter `offset` is used if the `coefficientMatrix` stores coefficients for multiple local bases at the same time. This happens for example for vector-valued test spaces. In this case, only the rows `offset` to `offset+` size of `testSearchSpace` of the `coefficientMatrix` are taken into account.

3.2.3 OptimalTestBasis

The near-optimal test space itself is implemented as a class called `OptimalTestBasis<TestspaceCoefficientMatrix, testIndex>`, which is compliant with the requirements for a global basis in the dune-module DUNE-FUNCTIONS [Engwer et al., 2015]. In the current implementation, `OptimalTestBasis` describes only one scalar variable. If there are more test variables, several `OptimalTestBasis` are needed. In this case, `testIndex` specifies the index of the component of the near-optimal test space which is described.

Like all global bases in DUNE-FUNCTIONS, the `OptimalTestBasis` provides a `LocalIndexSet` for the mapping of local to global degrees of freedom as well as a `LocalView` to provide access to all local basis functions whose support has non-trivial intersection with a given element. Since every optimal test basis function corresponds to a basis function in the trial spaces, the mapping of local to global degrees follows directly from the corresponding `LocalIndexSets` of the trial spaces. If there is more than one trial space, the global degrees of freedom are ordered by trial space, that is the first global degrees of freedom are those for the first trial space, the next ones are for the second trial space and so on. The `LocalView` of the `OptimalTestBasis` provides access to the correct `OptimalTestLocalFiniteElement`. To this end, in the method `bind(const Element& e)`, the coefficients for the local near-optimal test basis are computed by binding the `coefficientMatrix` to the element `e` and the corresponding `OptimalTestLocalFiniteElement` is constructed.

Note that for test spaces with more than one component, the computation of the coefficients for the local near-optimal test basis in general cannot be seperated for the different components. That is why the `TestspaceCoefficientMatrix` is implented as a seperate class so that several near-optimal test spaces can share one `TestspaceCoefficientMatrix` and the computed coefficients can be reused if the `TestspaceCoefficientMatrix` is bound several times to the same element by different near-optimal test bases.

The following lines of code show how to create an `OptimalTestBasis` consisting of two variables for a given `BilinearForm` and `InnerProduct`.

C++ code

```
TestspaceCoefficientMatrix testspaceCoefficientMatrix(bilinearForm, innerProduct);
Functions::OptimalTestBasis<TestspaceCoefficientMatrix, 0>
    feBasisTest0(testspaceCoefficientMatrix);
Functions::OptimalTestBasis<TestspaceCoefficientMatrix, 1>
    feBasisTest1(testspaceCoefficientMatrix);
```

3.3 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

$$\langle R_K(u_H, f), v\rangle_{\mathbb{V}_K} = b(u_H, v) - f(v), \quad \forall v \in \mathbb{V}_K. \tag{24}$$

Since (24) 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

$$\|\overline{R}(u_H, f)\|_{\mathbb{V}} := \left(\sum_{K\in\Omega_H} \|\overline{R}_K(u_H, f)\|^2_{\mathbb{V}_K}\right)^{1/2}. \tag{25}$$

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

In Dune-DPG, the computation of the a posteriori estimator (25) is handled by the class `ErrorTools`. The following lines of code compute (25) for the solution `u_H` of a problem with bilinear form `bilinearForm`, inner product `innerProduct` and right hand side `rhsVector`.

C++ code

```
ErrorTools errorTools = ErrorTools();
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`.

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

As a simple numerical example, we solve the transport problem (14) with

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

As Figure 2d shows, the exact solution φ 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. Ω_h is a refinement of Ω_H to some level $\ell \in \mathbb{N}_0$ such that $h = 2^{-\ell}H$.

With $\mathbb{U}_H$ and $\mathbb{V}_h$ defined as in (17) and (18) with $m = 2$, we compute $u_H = (\varphi_H, \theta_H) \in \mathbb{U}_H$ by solving the ultra-weak variational formulation (19). 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, $\forall K \in \Omega_H$.

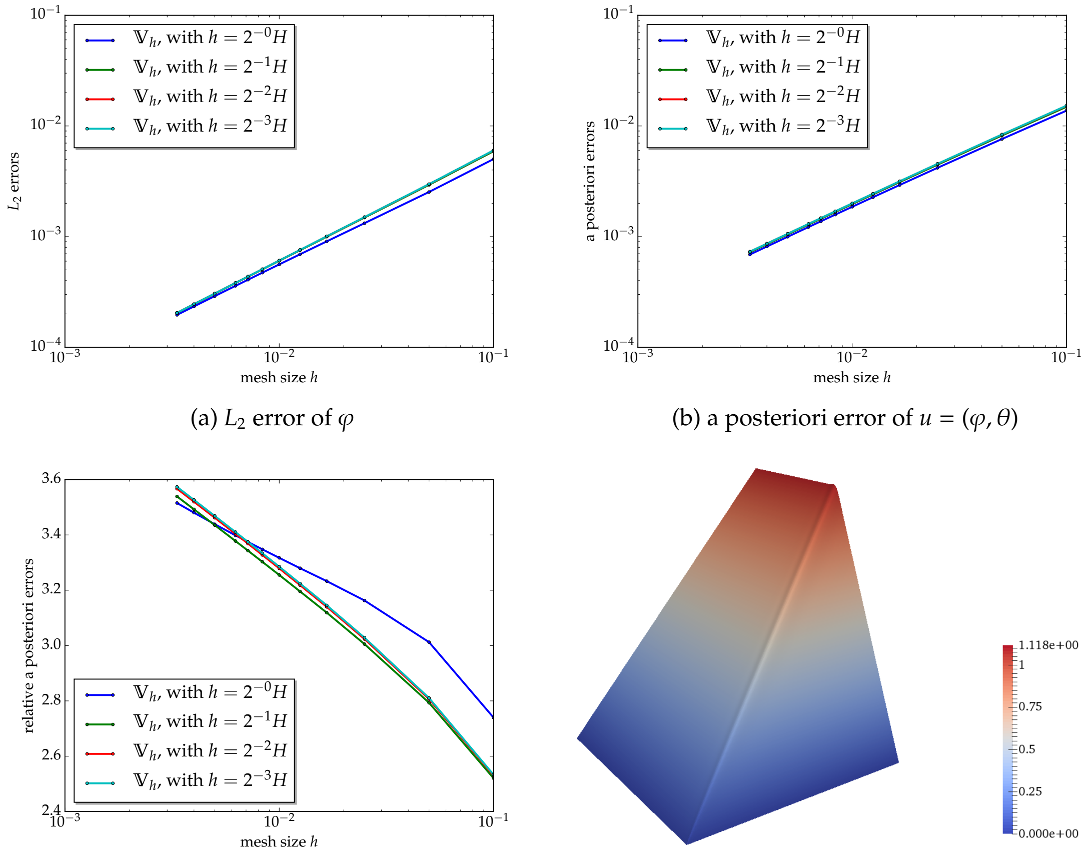
Regarding the error $\|\varphi - \varphi_H\|_{L_2(\Omega)}$, as Figure 2a shows, we observe linear convergence as H decreases. This is to be expected since the polynomial degree to compute φ_H is 1. The figure also shows that the refinement level ℓ of the test-search space $\mathbb{V}_h$ has essentially 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 2b and 2c 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 θ_H, namely $\|u - u_H\|_{\mathbb{U}} = \|(\varphi, \theta) - (\varphi_H, \theta_H)\|_{\mathbb{U}}$.

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 implement adaptive mesh refinements that would be driven by our local a posteriori error indicators.

Finally, 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 mayor 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



(a) L_2 error of φ

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

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

(d) Solution φ.

Figure 2: L_2 error and a posteriori error estimator of numerical solutions

that this would give us increased maintainability and decreased compile times in addition to the improvements in the usability of vector valued problems. This is aligned with our long-term goal of 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. 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

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*, 2016.

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. July 2015.

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*, 213–216:216–138, 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.

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.