A domain-specific embedded language in C++ for lowest-order discretizations of diffusive problems on general meshes

Daniele Antonio Di Pietro, Jean-Marc Gratien, Christophe Prud'Homme

A DOMAIN-SPECIFIC EMBEDDED LANGUAGE IN C++ FOR LOWEST-ORDER DISCRETIZATIONS OF DIFFUSIVE PROBLEMS ON GENERAL MESHES

DANIELE A. DI PIETRO*, JEAN-MARC GRATIEN†, AND CHRISTOPHE PRUD'HOMME‡

Abstract. In this work we propose an original implementation of a large family of lowest-order methods for diffusive problems including standard and hybrid finite volume methods, mimetic finite difference-type schemes, and cell centered Galerkin methods. The key idea is to regard the method at hand as a (Petrov–)Galerkin scheme based on possibly incomplete, broken affine spaces defined from a gradient reconstruction and a point value. The resulting unified framework serves as a basis for the development of a `FreeFEM`-like domain specific language targeted at defining discrete linear and bilinear forms. Both the back-end and the front-end of the language are extensively discussed, and several examples of applications are provided. The overhead of the language is evaluated by comparing with a more traditional implementation. A benchmark including the comparison with more classical finite element methods on standard meshes is also proposed.

Key words. Domain specific embedded language, finite volume methods, cell centered Galerkin methods, Petrov–Galerkin methods

AMS subject classifications. 65N08, 65N30, 65Y05

1. Introduction. Lowest-order methods possibly featuring conservation of physical quantities are traditionally employed in industrial applications where computational cost is a crucial issue. In this context, the use of general polyhedral, possibly nonconforming meshes commends itself for a number of reasons. To cite a few: (i) remeshing can be avoided or postponed in problems that involve mesh deformation — e.g. in sedimentary basin modeling non-standard elements and nonconformities can appear due to the erosion of geological layers; — (ii) the number of degrees of freedom can be reduced by aggregative coarsening techniques — cf. [7] for an application in the context of discontinuous Galerkin (dG) methods; — (iii) geometrical features can be represented more accurately without unduly increasing the number of mesh elements.

Handling general polyhedral meshes requires numerical schemes that possess the usual properties of stability and consistency. In the context of cell centered finite volume methods, a popular way to achieve consistency on general polyhedral meshes is provided by Multipoint Finite Volume schemes independently introduced by Aavatsmark, Barkve, Bøe and Mannseth [2] and Edwards and Rogers [22]. The main advantage of multipoint schemes is that they can be easily fitted into existing simulators based on standard finite volume schemes. A major drawback is their lack of stability in some configurations. Two ways of overcoming this difficulty by designing discretizations based on the variational formulation of the problem and featuring cell- and face-unknowns have been proposed by Brezzi, Lipnikov and coworkers [8, 9] (Mimetic Finite Difference methods) and by Droniou and Eymard [20] (Mixed/Hybrid Finite Volume methods). In this context, Eymard, Gallouët and Herbin [24] have shown that face unknowns can be selectively used as Lagrange multipliers to enforce flux continuity, or eliminated using a consistent interpolator (SUSHI scheme). More generally, this point of view leads to the notion that the discretization method can be locally adapted to the features of the problem. A different approach based on the analogy between lowest order methods in variational formulation and discontinuous Galerkin methods has been proposed by one of the authors in [13–15] (Cell Centered Galerkin methods). The key advantage of this approach is that it largely benefits from the well-established theory for discontinuous Galerkin methods [17].

When it comes to numerical performance, recent benchmarks [25, 30] have pointed out that the choice of the scheme for a given problem should be driven by multiple factors including, e.g.,

*IFP Energies nouvelles, 1 & 4 avenue Bois Préau, 92852 Rueil-Malmaison CEDEX, France, `dipietrd@ifpen.fr`, corresponding author

†IFP Energies nouvelles, 1 & 4 avenue Bois Préau, 92852 Rueil-Malmaison CEDEX, France, `j-marc.gratien@ifpen.fr`

‡Laboratoire Jean Kuntzmann, Université Joseph Fourier Grenoble 1, BP 53 38041 Grenoble Cedex 9, France,`christophe.prudhomme@ujf-grenoble.fr`

the features of the problem itself (heterogeneity, presence of convection), the computational mesh (which may result from an upstream modeling process), and the required precision. In this respect, there is an increasing urge to dispose of libraries covering a wide range of lowest-order methods and applications based on similar experiences in the context of Finite Element (FE) methods. Finite element libraries have nowadays reached a good level of maturity. Just to mention a few, we recall `Feel++` [33–35], `FEniCS` [31], `FreeFEM++` [29]. All of the above projects provide a user-friendly front-end in the form of a Domain Specific Language (DSL) possibly embedded in a general purpose, high-level hosting language (Domain Specific Embedded Language or DSEL). DS(E)Ls are an established means to break the complexity of applications by allowing each contributor to express themselves in a language as close as possible to their technical jargon. In the context of scientific computing, complexity spans different levels:

(i) *Modeling.* Modelers investigate more and more comprehensive physical models expressed in terms of (systems of) Partial Differential Equations (PDEs) possibly completed by algebraic closure laws;

(ii) *Discretization.* Numericians confront with an increasing number of discretization methods which are potentially suited to convert the PDE problem into a system of algebraic equations. Disposing of different discretization methods within a unified framework is highly beneficial since it allows to identify the most efficient choice for the problem at hand;

(iii) *Solution.* Several low-level numerical packages are available to solve systems of algebraic equations. Their performance in terms of computational efficiency and stability is strongly related to both the features of the matrix to solve (symmetry, fill-in pattern, etc.) and to the underlying hardware architecture;

(iv) *Software design.* Finally, computer scientists design low-level data structures and algorithm that benefit from the evolution of both hardware architectures and languages to ensure the overall efficiency.

Ideally, a software platform should allow contributors at each level to focus on a specific aspect of the problem without being hindered by the interaction with the other levels. The ultimate goal of this work is to develop a DSEL aimed at numericians that allow to express a large family of lowest-order using a syntax as close as possible to the mathematical notation.

An instrumental step in this direction is to provide a unified framework for the methods cited above, which we do in Sect. 2. The key idea is here to reformulate the method at hand as a (Petrov–)Galerkin scheme based on possibly incomplete broken affine spaces. More specifically, a method is defined in three steps by prescribing:

(i) *The degrees of freedom.* Degrees Of Freedom (DOFs) are attached to mesh items such as cells, faces, or nodes and ultimately represent the unknowns of the problems;

(ii) *The discrete function space(s).* Starting from the prescribed set of degrees of freedom, we identify a piecewise constant gradient reconstruction from which piecewise affine basis functions are defined. The function space spanned by the basis functions is often incomplete in the sense that it allows to represent only a subset of the broken affine functions on the mesh;

(iii) *The formulation.* The method (or a linearized version thereof) is defined by linear and bilinear forms acting on the discrete function spaces defined in the previous point.

This unified perspective, drawing on the lines of [15] and of the precursor work [5], allows, in particular, to recycle many ideas originally developed in the context of FE methods. A major difference is that the reconstructed differential operators often have unconventional stencils, possibly spanning neighbor elements and depending on problem data such as the diffusion coefficient. As a consequence, the classical FE approach based on a reference element (see, e.g., [23, Chap. 7–8]) proves inadequate.

The implementation in `C++` is described in detail in Sect. 3 and uses the latest standard `C++-11`. As discussed earlier in this section, the DSEL acts as an interface between the numerician, who focuses on discretization methods, and the computer scientist, who manages low-level services such as parallelism and input/output services. As a result, the exposition addresses two levels:

(i) *The back-end.* This is the foundation for the user-friendly interface, and it contains the programming counterpart of vector spaces of DOFs as well as an original implementation of linear operators with unconventional stencil. Although the tools developed at this stage

would allow to implement all of the methods discussed in Sect. 2, this would require to cope directly with low-level computer services;

(ii) *The front-end.* To avoid this, the front-end provides a specific language which conceals most of the implementation details and allows the numerician to focus on discretization methods. Although the DSEL discussed in this section is closely inspired by `Feel++` [33], a few modifications are proposed such as a different treatment for test and trial functions and index notation for vector problems.

In Sect. 4 we provide several numerical examples to assess the performance of the proposed approach. A special care is devoted to the evaluation of the overhead of the DSEL and to the comparison with more standard methods/implementations. The actual implementation of the DSEL is based on the `Arcane` framework [27], a proprietary platform conjointly developed at CEA-DAM and IFP Energies nouvelles which handles technical aspects such as memory management, parallelism and post-processing.

2. Setting. In this section we introduce a unified framework for lowest-order methods closely inspired by [19]. The key idea is here to reformulate the method at hand as a (Petrov–)Galerkin scheme based on a possibly incomplete, broken affine space. This is done by introducing a piecewise constant gradient reconstruction, which is used to recover a broken affine function starting from cell (and possibly face) unknowns. The material is organized as follows. In Sect. 2.1 we present the discrete setting including the main notations. In Sect. 2.2 we introduce incomplete piecewise affine broken spaces, which are used in Sect. 2.3 to formulate some popular lowest-order methods. In Sect. 2.4 we discuss an application to the Stokes problem.

2.1. Discretization. Let $\Omega \subset \mathbb{R}^d$, $d \geq 2$, denote a bounded connected polyhedral domain. The first ingredient in the definition of lowest-order methods is a suitable discretization of Ω. We denote by $\mathcal{T}_h$ a finite collection of nonempty, disjoint open polyhedra $\mathcal{T}_h = \{T\}$ forming a partition of Ω such that $h = \max_{T\in\mathcal{T}_h} h_T$, with h_T denoting the diameter of the element $T \in \mathcal{T}_h$. Admissible meshes include general polyhedral discretizations with possibly nonconforming interfaces; see Figure 2.1 for an example in $d = 2$. Mesh nodes are collected in the set $\mathcal{N}_h$ and, for all $T \in \mathcal{T}_h$, $\mathcal{N}_T$ contains the nodes that lie on the boundary of T. We say that a hyperplanar closed subset F of $\overline{\Omega}$ is a mesh face if it has positive $(d-1)$-dimensional measure and if either there exist T_1, $T_2 \in \mathcal{T}_h$ such that $F \subset \partial T_1 \cap \partial T_2$ (and F is called an *interface*) or there exists $T \in \mathcal{T}_h$ such that $F \subset \partial T \cap \partial\Omega$ (and F is called a *boundary face*). Interfaces are collected in the set $\mathcal{F}_h^i$, boundary faces in $\mathcal{F}_h^b$ and we let $\mathcal{F}_h := \mathcal{F}_h^i \cup \mathcal{F}_h^b$. Moreover, we set, for all $T \in \mathcal{T}_h$,

$$\mathcal{F}_T := \{F \in \mathcal{F}_h \mid F \subset \partial T\}. \tag{2.1}$$

Symmetrically, for all $F \in \mathcal{F}_h$, we define

$$\mathcal{T}_F := \{T \in \mathcal{T}_h \mid F \subset \partial T\}.$$

The set $\mathcal{T}_F$ consists of exactly two mesh elements if $F \in \mathcal{F}_h^i$ and of one if $F \in \mathcal{F}_h^b$. For all mesh nodes $P \in \mathcal{N}_h$, $\mathcal{F}_P$ denotes the set of mesh faces sharing P, i.e.

$$\mathcal{F}_P := \{F \in \mathcal{F}_h \mid P \in F\}. \tag{2.2}$$

For every interface $F \in \mathcal{F}_h^i$ we introduce an arbitrary but fixed ordering of the elements in $\mathcal{T}_F$ and let $\mathrm{n}_F = \mathrm{n}_{T_1,F} = -\mathrm{n}_{T_2,F}$, where $\mathrm{n}_{T_i,F}$, $i \in \{1,2\}$, denotes the unit normal to F pointing out of $T_i \in \mathcal{T}_F$. On a boundary face $F \in \mathcal{F}_h^b$, n_F denotes the unit normal pointing out of Ω. The barycenter of a face $F \in \mathcal{F}_h$ is denoted by $\overline{\mathbf{x}}_F := \int_F \mathbf{x}/|F|$.

For each element $T \in \mathcal{T}_h$ we identify a point $\mathbf{x}_T \in T$ (the *cell center*) such that T is star-shaped with respect to $\mathbf{x}_T$. For all $F \in \mathcal{F}_T$ we let

$$d_{T,F} := \operatorname{dist}(\mathbf{x}_T, F).$$

It is assumed that, for all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$, $d_{T,F} > 0$ is uniformly comparable to h_T. Starting from cell centers we can define a pyramidal submesh of $\mathcal{T}_h$ as follows:

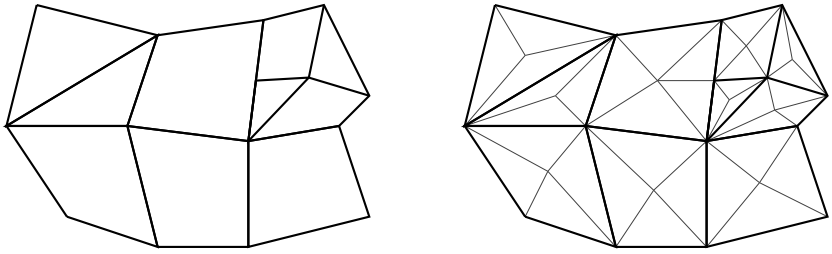


FIG. 2.1. *Mesh $\mathcal{T}_h$ (*left*) and pyramidal submesh $\mathcal{P}_h$ (*right*)*

$$\mathcal{P}_h := \{\mathcal{P}_{T,F}\}_{T\in\mathcal{T}_h,\, F\in\mathcal{F}_T},$$

where, for all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$, $\mathcal{P}_{T,F}$ denotes the open pyramid of apex $\mathbf{x}_T$ and base F, i.e.,

$$\mathcal{P}_{T,F} := \{\mathbf{x} \in T \mid \exists \mathbf{y} \in F \setminus \partial F,\, \exists \theta \in (0,1) \mid \mathbf{x} = \theta\mathbf{y} + (1-\theta)\mathbf{x}_T\}.$$

The pyramids $\{\mathcal{P}_{T,F}\}_{T\in\mathcal{T}_h,\, F\in\mathcal{F}_T}$ are nondegenerate by assumption. Let $\mathcal{S}_h$ be such that

$$\mathcal{S}_h = \mathcal{T}_h \text{ or } \mathcal{S}_h = \mathcal{P}_h. \tag{2.3}$$

For all $k \geq 0$, we define the broken polynomial spaces of total degree $\leq k$ on $\mathcal{S}_h$,

$$\mathbb{P}_d^k(\mathcal{S}_h) := \{v \in L^2(\Omega) \mid \forall S \in \mathcal{S}_h,\, v|_S \in \mathbb{P}_d^k(S)\},$$

with $\mathbb{P}_d^k(S)$ given by the restriction to $S \in \mathcal{S}_h$ of the functions in $\mathbb{P}_d^k$.

REMARK 2.1 (Admissible mesh sequence). *In the context of* a priori *convergence analysis for vanishing mesh size h it is necessary to bound some quantities uniformly with respect to h. This leads to the concept of* admissible mesh sequence*. This topic is not addressed in detail herein since our focus is mainly on implementation. For a comprehensive discussion we refer to Brezzi, Lipnikov and coworkers [8, 9], Droniou and Eymard [20], Eymard, Gallouët, and Herbin [24], Di Pietro and Ern [17, Chap. 1] and Di Pietro [15].*

We close this section by introducing trace operators which are of common use in the context of nonconforming FE methods. Let v be a scalar-valued function defined on Ω smooth enough to admit on all $F \in \mathcal{F}_h$ a possibly two-valued trace. To any interface $F \subset \partial T_1 \cap \partial T_2$ we assign two non-negative real numbers $\omega_{T_1,F}$ and $\omega_{T_2,F}$ such that

$$\omega_{T_1,F} + \omega_{T_2,F} = 1,$$

and define the jump and weighted average of v at F for a.e. $\mathbf{x} \in F$ as

$$[\![v]\!]_F(\mathbf{x}) := v|_{T_1} - v|_{T_2}, \qquad \{v\}_{\omega,F}(\mathbf{x}) := \omega_{T_1,F} v|_{T_1}(\mathbf{x}) + \omega_{T_2,F} v|_{T_2}(\mathbf{x}). \tag{2.4}$$

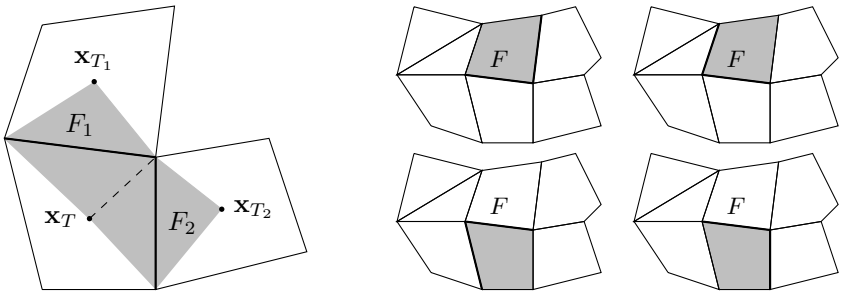
If $F \in \mathcal{F}_h^b$ with $F = \partial T \cap \partial\Omega$, we conventionally set $\{v\}_{\omega,F}(\mathbf{x}) = [\![v]\!]_F(\mathbf{x}) = v|_T(\mathbf{x})$. The index ω is omitted from the average operator when $\omega_{T_1,F} = \omega_{T_2,F} = \frac{1}{2}$, and we simply write $\{v\}_F(\mathbf{x})$. The dependence on both the point $\mathbf{x}$ and the face F is also omitted from both the jump and average trace operators if no ambiguity arises.

2.2. A unified abstract perspective for lowest-order methods. The key idea to get a unifying perspective is to consider lowest-order methods as nonconforming methods based on incomplete broken affine spaces that are defined starting from the space of degrees of freedom (DOFs) $\mathbb{V}_h$. More precisely, we let

$$\mathbb{T}_h := \mathbb{R}^{\mathcal{T}_h}, \qquad \mathbb{F}_h := \mathbb{R}^{\mathcal{F}_h},$$

and consider the following choices:

$$\mathbb{V}_h = \mathbb{T}_h \text{ or } \mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h. \tag{2.5}$$



(a) Set $\mathcal{T}_\mathfrak{g} = \{T, T_1, T_2\}$ for the L-group $\mathfrak{g} = \{F_1, F_2\}$

(b) $\mathcal{G}_F$ = L-groups containing the face F

FIG. 2.2. *L-construction*

The choice $\mathbb{V}_h = \mathbb{T}_h$ corresponds to cell centered finite volume (CCFV) and cell centered Galerkin (ccG) methods, while the choice $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$ leads to mimetic finite difference (MFD) and mixed/hybrid finite volume (MHFV) methods. The key ingredient in the definition of a broken affine space is a piecewise constant linear gradient reconstruction $\mathfrak{G}_h : \mathbb{V}_h \to [\mathbb{P}_d^0(\mathcal{S}_h)]^d$ with suitable properties. We emphasize that the linearity of $\mathfrak{G}_h$ is a foundamental assumption for the implementation discussed in Sect. 3.

Using the above ingredients, we can define the linear operator $\mathfrak{R}_h : \mathbb{V}_h \to \mathbb{P}_d^1(\mathcal{S}_h)$ such that, for all $\mathbf{v}_h \in \mathbb{V}_h$,

$$\forall S \in \mathcal{S}_h,\, S \subset T_S \in \mathcal{T}_h,\, \forall \mathbf{x} \in S, \qquad \mathfrak{R}_h(\mathbf{v}_h)|_S = v_{T_S} + \mathfrak{G}_h(\mathbf{v}_h)|_S \cdot (\mathbf{x} - \mathbf{x}_{T_S}). \tag{2.6}$$

The operator $\mathfrak{R}_h$ maps every vector of DOFs $\mathbf{v}_h \in \mathbb{V}_h$ onto a piecewise affine function $\mathfrak{G}_h(\mathbf{v}_h)$ belonging to $\mathbb{P}_d^1(\mathcal{S}_h)$. Hence, we can define a broken affine space as follows:

$$V_h = \mathfrak{R}_h(\mathbb{V}_h) \subset \mathbb{P}_d^1(\mathcal{S}_h). \tag{2.7}$$

The operator $\mathfrak{R}_h$ is assumed to be injective, so that a bijective operator can be obtained by restricting its codomain. In what follows we show how some common lowest-order methods can be interpreted in these terms. To simplify the exposition, we focus on discrete spaces approximating $H_0^1(\Omega)$, i.e., possibly including strongly enforced boundary conditions.

2.3. Pure diffusion. In this section we provide a few examples for the gradient operator $\mathfrak{G}_h$ that allow to recover some of the methods listed in the previous section for the following heterogeneous diffusion model problem:

$$\begin{aligned} -\nabla\cdot(\kappa\nabla u) &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned}$$

with source term $f \in L^2(\Omega)$. Here, κ denotes a uniformly elliptic tensor field piecewise constant on the mesh $\mathcal{T}_h$.

2.3.1. The G-method. As a first example we consider the special instance of CCFV methods analyzed in [4]. A preliminary step consists in introducing the so-called L-construction originally proposed by Aavatsmark, Eigestad, Mallison, and Nordbotten [3]. The key idea of the L-construction is to use $d+1$ cell and boundary face values (provided, in this case, by the homogeneous boundary condition) to express a continuous piecewise affine function with continuous diffusive fluxes. The values are selected using d neighboring faces belonging to a cell and sharing a common vertex. More precisely, we define the set of L-groups as follows:

$$\mathcal{G} := \{\mathfrak{g} \subset \mathcal{F}_T \cap \mathcal{F}_P,\, T \in \mathcal{T}_h,\, P \in \mathcal{N}_T \mid \operatorname{card}(\mathfrak{g}) = d\},$$

with $\mathcal{F}_T$ and $\mathcal{F}_P$ given by (2.1) and (2.2) respectively. It is useful to introduce a symbol for the set of cells concurring in the L-construction: For all $\mathfrak{g} \in \mathcal{G}$, we let (cf. Figure 2.2(a))

$$\mathcal{T}_\mathfrak{g} := \{T \in \mathcal{T}_h \mid T \in \mathcal{T}_F,\, F \in \mathfrak{g}\}.$$

Let now $\mathfrak{g} \in \mathcal{G}$ and denote by $T_\mathfrak{g}$ an element $T_\mathfrak{g}$ such that $\mathfrak{g} \subset \mathcal{F}_{T_\mathfrak{g}}$ (this element may not be unique). For all $\mathbf{v}_h \in \mathbb{V}_h$ we construct the function $\xi^\mathfrak{g}_{\mathbf{v}_h}$ piecewise affine on the family of pyramids $\{\mathcal{P}_{T,F}\}_{F\in\mathfrak{g},\,T\in\mathcal{T}_\mathfrak{g}}$ such that: (i) $\xi^\mathfrak{g}_{\mathbf{v}_h}(\mathbf{x}_T) = v_T$ for all $T \in \mathcal{T}_\mathfrak{g}$ and $\xi^\mathfrak{g}_{\mathbf{v}_h}(\overline{\mathbf{x}}_F) = 0$ for all $F \in \mathfrak{g} \cap \mathcal{F}_h^b$; (ii) $\xi^\mathfrak{g}_{\mathbf{v}_h}$ is affine inside $T_\mathfrak{g}$ and is continuous across every interface in the group: For all $F \in \mathfrak{g} \cap \mathcal{F}_h^i$ such that $F \subset \partial T_1 \cap \partial T_2$,

$$\forall \mathbf{x} \in F, \qquad \xi^\mathfrak{g}_{\mathbf{v}_h}|_{T_1}(\mathbf{x}) = \xi^\mathfrak{g}_{\mathbf{v}_h}|_{T_2}(\mathbf{x});$$

(iii) $\xi^\mathfrak{g}_{\mathbf{v}_h}$ has continuous diffusive flux across every interface in the group: For all $F \in \mathfrak{g} \cap \mathcal{F}_h^i$ such that $F \subset \partial T_1 \cap \partial T_2$,

$$(\kappa\nabla\xi^\mathfrak{g}_{\mathbf{v}_h})|_{T_1}\cdot\mathrm{n}_F = (\kappa\nabla\xi^\mathfrak{g}_{\mathbf{v}_h})|_{T_2}\cdot\mathrm{n}_F.$$

For further details on the L-construction including an explicit formula for $\xi^\mathfrak{g}_{\mathbf{v}_h}$ we refer to [3, 4]. For every face $F \in \mathcal{F}_h$ we define the set $\mathcal{G}_F$ of L-groups containing F (see Figure 2.2(b)),

$$\mathcal{G}_F := \{\mathfrak{g} \in \mathcal{G} \mid F \in \mathfrak{g}\}, \tag{2.8}$$

and introduce the set of nonnegative weights $\{\varsigma_{\mathfrak{g},F}\}_{\mathfrak{g}\in\mathcal{G}_F}$ such that $\sum_{\mathfrak{g}\in\mathcal{G}_F}\varsigma_{\mathfrak{g},F} = 1$. The trial space for the G-method is obtained as follows: (i) let $\mathcal{S}_h = \mathcal{P}_h$ and $\mathbb{V}_h = \mathbb{T}_h$; (ii) let $\mathfrak{G}_h = \mathfrak{G}_h^\mathrm{g}$ with $\mathfrak{G}_h^\mathrm{g}$ such that

$$\forall \mathbf{v}_h \in \mathbb{T}_h,\, \forall T \in \mathcal{T}_h,\, \forall F \in \mathcal{F}_T, \qquad \mathfrak{G}_h^\mathrm{g}(\mathbf{v}_h)|_{\mathcal{P}_{T,F}} = \sum_{\mathfrak{g}\in\mathcal{G}_F}\varsigma_{\mathfrak{g},F}\nabla\xi^\mathfrak{g}_{\mathbf{v}_h}|_{\mathcal{P}_{T,F}}. \tag{2.9}$$

We denote by $\mathfrak{R}_h^\mathrm{g}$ the reconstruction operator defined as in (2.6) with $\mathfrak{G}_h = \mathfrak{G}_h^\mathrm{g}$ and let $V_h^\mathrm{g} := \mathfrak{R}_h^\mathrm{g}(\mathbb{V}_h)$. The G-method of [4] is then equivalent to the following Petrov–Galerkin method:

$$\text{Find } u_h \in V_h^\mathrm{g} \text{ s.t. } a_h^\mathrm{g}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in \mathbb{P}_d^0(\mathcal{T}_h), \tag{2.10}$$

where $a_h^\mathrm{g}(u_h, v_h) := -\sum_{F\in\mathcal{F}_h}\int_F\{\kappa\nabla_h u_h\}\cdot\mathrm{n}_F[\![v_h]\!]$ with ∇_h broken gradient on $\mathcal{S}_h$.

Remark 2.2 (An unconditionally stable method). *The main drawback of the G-method is that stability can only be proved under quite stringent conditions; see, e.g., [4, Lemma 3.4]. A possible way to circumvent this difficulty has been recently proposed by one of the authors [14] in the context of ccG methods. The key idea is to use V_h^g both as a trial and test space, and modify the discrete bilinear form to recover both consistency and stability. Since the discrete functions in V_h^g are discontinuous across the lateral faces of the pyramids in $\mathcal{P}_h$, least-square penalization of the jumps is required to assert stability in terms of coercivity. The resulting method also enters the present framework, but is not detailed here for the sake of conciseness.*

2.3.2. A cell centered Galerkin method. The L-construction is used to define a trace reconstruction in the ccG method of [13,15]. More specifically, for all $F \in \mathcal{F}_h^i$, we select one group $\mathfrak{g}_F \in \mathcal{G}_F$ with $\mathcal{G}_F$ defined by (2.8) and introduce the linear trace operator $\mathbf{T}_h^\mathrm{g} : \mathbb{T}_h \to \mathbb{F}_h$ which maps every vector of cell centered DOFs $\mathbf{v}_h \in \mathbb{T}_h$ on a vector $(v_F)_{F\in\mathcal{F}_h} \in \mathbb{F}_h$ such that

$$v_F = \begin{cases} \xi^{\mathfrak{g}_F}_{\mathbf{v}_h}(\overline{\mathbf{x}}_F) & \text{if } F \in \mathcal{F}_h^i, \\ 0 & \text{if } F \in \mathcal{F}_h^b. \end{cases} \tag{2.11}$$

The trace operator $\mathbf{T}_h^\mathrm{g}$ is then employed in a gradient reconstruction based on Green's formula and inspired by Eymard, Gallouët, and Herbin [24]. More precisely, we introduce the linear gradient operator $\mathfrak{G}_h^\mathrm{green} : \mathbb{T}_h \times \mathbb{F}_h \to [\mathbb{P}_d^0(\mathcal{T}_h)]^d$ such that, for all $(\mathbf{v}^\mathcal{T}, \mathbf{v}^\mathcal{F}) \in \mathbb{T}_h \times \mathbb{F}_h$ and all $T \in \mathcal{T}_h$,

$$\mathfrak{G}_h^\mathrm{green}(\mathbf{v}^\mathcal{T}, \mathbf{v}^\mathcal{F})|_T = \frac{1}{|T|}\sum_{F\in\mathcal{F}_T}|F|(v_F - v_T)\mathrm{n}_{T,F}. \tag{2.12}$$

The discrete space for the ccG method under examination can then be obtained as follows: (i) let $\mathcal{S}_h = \mathcal{T}_h$ and $\mathbb{V}_h = \mathbb{T}_h$; (ii) let $\mathfrak{G}_h = \mathfrak{G}_h^{\mathrm{ccg}}$ with $\mathfrak{G}_h^{\mathrm{ccg}}$ such that

$$\forall \mathbf{v}_h \in \mathbb{V}_h, \qquad \mathfrak{G}_h^{\mathrm{ccg}}(\mathbf{v}_h) = \mathfrak{G}_h^{\mathrm{green}}(\mathbf{v}_h, \mathbf{T}_h^{\mathrm{g}}(\mathbf{v}_h)). \tag{2.13}$$

The reconstruction operator defined taking $\mathfrak{G}_h = \mathfrak{G}_h^{\mathrm{ccg}}$ in (2.6) is denoted by $\mathfrak{R}_h^{\mathrm{ccg}}$, and the corresponding discrete space by $V_h^{\mathrm{ccg}} := \mathfrak{R}_h^{\mathrm{ccg}}(\mathbb{T}_h)$. We define the weights in the average operator as follows: For all $F \in \mathcal{F}_h^i$ such that $F \subset \partial T_1 \cap \partial T_2$,

$$\omega_{T_1,F} = \tfrac{\lambda_2}{\lambda_1+\lambda_2}, \qquad \omega_{T_2,F} = \tfrac{\lambda_1}{\lambda_1+\lambda_2},$$

where $\lambda_i := \kappa|_{T_i}\mathrm{n}_F\cdot\mathrm{n}_F$ for $i \in \{1,2\}$. Set, for all $(u_h, v_h) \in V_h^{\mathrm{ccg}} \times V_h^{\mathrm{ccg}}$,

$$\begin{aligned} a_h^{\mathrm{ccg}}(u_h, v_h) := & \int_\Omega \kappa\nabla_h u_h\cdot\nabla_h v_h - \sum_{F\in\mathcal{F}_h}\int_F [\{\kappa\nabla_h u_h\}_\omega\cdot\mathrm{n}_F[\![v_h]\!] + [\![u_h]\!]\{\kappa\nabla v_h\}_\omega\cdot\mathrm{n}_F] \\ & + \sum_{F\in\mathcal{F}_h}\eta\frac{\gamma_F}{h_F}\int_F [\![u_h]\!][\![v_h]\!], \end{aligned} \tag{2.14}$$

with ∇_h broken gradient on $\mathcal{T}_h$, $\gamma_F = \frac{2\lambda_1\lambda_2}{\lambda_1+\lambda_2}$ on internal faces $F \subset \partial T_1 \cap \partial T_2$, $\gamma_F = \kappa|_T\mathrm{n}_F\cdot\mathrm{n}_F$ on boundary faces $F \subset \partial T \cap \partial\Omega$ and η is a (strictly positive) penalty parameter. The ccG method reads

$$\text{Find } u_h \in V_h^{\mathrm{ccg}} \text{ s.t. } a_h^{\mathrm{ccg}}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in V_h^{\mathrm{ccg}}. \tag{2.15}$$

The bilinear form a_h^{ccg} has been originally introduced by Di Pietro, Ern and Guermond [18] in the context of dG methods for degenerate advection-diffusion-reaction problems. In particular, when the diffusion field κ is homogeneous, the method (2.15) coincides with the Symmetric Interior Penalty (SIP) method of Arnold [6] associated to the bilinear form

$$\begin{aligned} a_h^{\mathrm{sip}}(u_h, v_h) = & \int_\Omega \kappa\nabla_h u_h\cdot\nabla_h v_h - \sum_{F\in\mathcal{F}_h}\int_F [\{\kappa\nabla_h u_h\}\cdot\mathrm{n}_F[\![v_h]\!] + [\![u_h]\!]\{\kappa\nabla_h v_h\}\cdot\mathrm{n}_F] \\ & + \sum_{F\in\mathcal{F}_h}\eta\frac{\gamma_F}{h_F}\int_F [\![u_h]\!][\![v_h]\!]. \end{aligned} \tag{2.16}$$

For further details on the link between ccG and discontinuous Galerkin methods we refer to [13–15].

Remark 2.3 (Numerical integration). *Recalling that the gradient reconstruction is piecewise constant one realizes that the integrals appearing in the first line of* (2.14) *can be evaluated exactly using the barycenter of the mesh item (cell or face) as a quadrature node. The penalty term in the second line involves the face integral of a quadratic polynomial, which would require a cubature with degree of exactness of (at least) 2. In practice, however, it suffices to penalize the low-order part of the jumps by replacing the penalty term in the second line of* (2.14) *by*

$$\sum_{F\in\mathcal{F}_h}\eta\frac{\gamma}{h_F}\int_F \langle[\![u_h]\!]\rangle_F\langle[\![v_h]\!]\rangle_F,$$

where $\langle\psi\rangle_F := \int_F \psi/|F|$. *For a discussion on penalty strategies acting on the low-degree part of the jumps we refer to [10, 28].*

2.3.3. The SUSHI method. As a last example we consider two variants of the SUSHI scheme of Eymard, Gallouët, and Herbin [24]; see also Droniou, Eymard, Gallouët, and Herbin [21] for a discussion on the link with the MFD methods of Brezzi, Lipnikov, and coworkers [8,9]. This method is based on the gradient reconstruction (2.12), but stabilization is achieved in a rather

different manner with respect to (2.14). More precisely, we define the linear residual operator $\mathfrak{r}_h : \mathbb{T}_h \times \mathbb{F}_h \to \mathbb{P}_d^0(\mathcal{P}_h)$ as follows: For all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$,

$$\mathfrak{r}_h(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{\mathcal{P}_{T,F}} = \frac{d^{\frac{1}{2}}}{d_{T,F}} \left[v_F - v_T - \mathfrak{G}_h^{\text{green}}(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_T \cdot (\overline{\mathbf{x}}_F - \mathbf{x}_T) \right].$$

We observe, in passing, that the factor $d^{\frac{1}{2}}$ can in general be replaced by a user-defined stabilization parameter $\eta > 0$. The advantage of taking $\eta = d^{\frac{1}{2}}$ is that it yields the classical two-point method on κ-orthogonal meshes. The discrete space for the SUSHI method with hybrid unknowns is obtained as follows: (i) let $\mathcal{S}_h = \mathcal{P}_h$ and $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$; (ii) let $\mathfrak{G}_h = \mathfrak{G}_h^{\text{hyb}}$ with $\mathfrak{G}_h^{\text{hyb}}$ such that, for all $(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}}) \in \mathbb{T}_h \times \mathbb{F}_h$, all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$,

$$\mathfrak{G}_h^{\text{hyb}}(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{\mathcal{P}_{T,F}} = \mathfrak{G}_h^{\text{green}}(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_T + \mathfrak{r}_h(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{\mathcal{P}_{T,F}} \mathrm{n}_{T,F}. \tag{2.17}$$

Denote by $\mathfrak{R}_h^{\text{hyb}}$ the reconstruction operator defined by (2.6) with $\mathfrak{G}_h = \mathfrak{G}_h^{\text{hyb}}$. The SUSHI method with hybrid unknowns reads

$$\text{Find } u_h \in V_h^{\text{hyb}} \text{ s.t. } a_h^{\text{sushi}}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in V_h^{\text{hyb}}, \tag{2.18}$$

with

$$a_h^{\text{sushi}}(u_h, v_h) := \int_\Omega \kappa \nabla_h u_h \cdot \nabla_h v_h, \tag{2.19}$$

and ∇_h broken gradient on $\mathcal{P}_h$. Alternatively, one can obtain a cell centered version by setting $\mathbb{V}_h = \mathbb{T}_h$ and replacing $\mathfrak{G}_h^{\text{hyb}}$ defined by (2.17) by $\mathfrak{G}_h = \mathfrak{G}_h^{\text{cc}}$ with $\mathfrak{G}_h^{\text{cc}}$ such that

$$\forall \mathbf{v}_h \in \mathbb{T}_h, \qquad \mathfrak{G}_h^{\text{cc}}(\mathbf{v}_h) = \mathfrak{G}_h^{\text{hyb}}(\mathbf{v}_h, \mathbf{T}_h^{\text{g}}(\mathbf{v}_h)), \tag{2.20}$$

and $\mathbf{T}_h^{\text{g}}$ defined by (2.11). This variant coincides with the version proposed in [24] for homogeneous κ, but it has the advantage to reproduce piecewise affine solutions to (2.3) on $\mathcal{T}_h$ when κ is heterogeneous. The discrete space obtained taking $\mathfrak{G}_h = \mathfrak{G}_h^{\text{cc}}$ in (2.7) is labeled V_h^{cc}.

2.4. Stokes. To close this section, we discuss a more complicated example involving a system of PDEs. More specifically, we focus on the steady Stokes problem

$$-\triangle u + \nabla p = f \qquad \text{in } \Omega, \tag{2.21a}$$
$$\nabla \cdot u = 0 \qquad \text{in } \Omega, \tag{2.21b}$$
$$u = 0 \qquad \text{on } \partial\Omega, \tag{2.21c}$$
$$\int_\Omega p = 0, \tag{2.21d}$$

where $u : \Omega \to \mathbb{R}^d$ is the vector-valued velocity field, $p : \Omega \to \mathbb{R}$ is the pressure, and $f : \Omega \to \mathbb{R}^d$ is the forcing term. Equations (2.21a) and (2.21b) express the conservation of momentum and mass respectively. The problem is supplemented by the homogeneous boundary condition (2.21c) (for a discussion on other boundary conditions we refer, e.g., to Ern and Guermond [23]). Following [15], we consider a discretization based on the spaces

$$U_h := [V_h^{\text{ccg}}]^d, \qquad P_h := \mathbb{P}_d^0(\mathcal{T}_h)/\mathbb{R}. \tag{2.22}$$

The momentum diffusion is discretized by the bilinear form $a_h \in \mathcal{L}(U_h \times U_h, \mathbb{R})$ such that

$$a_h(u_h, v_h) = \sum_{i=1}^d a_h^{\text{sip}}(u_{h,i}, v_{h,i}), \tag{2.23}$$

where, for all $w_h \in U_h$, the Cartesian components of w_h are denoted by $(w_{h,i})_{i\in\{1,\dots,d\}}$. The velocity-pressure coupling hinges on the bilinear form $b_h \in \mathcal{L}(U_h \times P_h, \mathbb{R})$ (see, e.g., [12]):

$$b_h(v_h, q_h) = -\int_\Omega (\nabla_h \cdot v_h) q_h + \sum_{F\in\mathcal{F}_h} \int_F [\![v_h]\!] \cdot \mathrm{n}_F \{q_h\}. \tag{2.24}$$

The discrete divergence operator associated to b_h is not surjective with choice of spaces (2.22). The stability of the velocity-pressure coupling can be recovered by penalizing pressure jumps via the bilinear form $s_h \in \mathcal{L}(P_h \times P_h, \mathbb{R})$ such that

$$s_h(p_h, q_h) = \sum_{F\in\mathcal{F}_h^i} \int_F h_F [\![p_h]\!][\![q_h]\!]. \tag{2.25}$$

The discrete problem reads: Find $(u_h, p_h) \in U_h \times P_h$ such that, for all $(v_h, q_h) \in U_h \times P_h$,

$$a_h(u_h, v_h) + b_h(v_h, p_h) - b_h(u_h, q_h) + s_h(p_h, q_h) = \int_\Omega f \cdot v_h. \tag{2.26}$$

3. Implementation. The framework of Sect. 2 serves as a basis for a DSEL targeted at expressing linear and bilinear forms using a syntax closely inspired by that of `Feel++` [34, 35]. To illustrate the capabilities of the DSEL in a nutshell, compare Listing 1 with the expression of the bilinear form a_h^{sip} (2.16).

LISTING 1
Implementation of the bilinear form a_h^{sip} defined by (2.16) using the DSEL. For the sake of simplicity, κ is assumed to be scalar-valued and homogeneous

```
// The space V_h^ccg spans functions in P_d^1(T_h) whose gradient is given by (2.13)
Mesh Th(/* ... */);
FunctionSpace<Mesh,
              span< poly<1>, gradient<GreenGradient> >
              >::type Vh(Th);
// Trial and test functions
auto uh = Vh.trialFunction("uh");
auto vh = Vh.testFunction("vh");
// Penalty parameter (eta and kappa are positive scalars)
auto penalty = eta*K/H();
// Bilinear form a_h^sip and linear form b
BilinearForm ah = integrate(allCells(Th),dot(K*grad(uh),grad(vh)))
                + integrate(allFaces(Th),
                            - jump(uh)*dot(N(),avg(K*grad(vh)))
                            - dot(N(),avg(K*grad(uh)))*jump(vh)
                            + penalty*jump(uh)*jump(vh)
                            );
LinearForm bh = integrate(allCells(Th), f*vh);
// Initialize context with global matrix A and global right-hand side vector b
LinearSystemContext ctx(A, b);
// Linear system assembly
eval(ah, ctx);
eval(bh, ctx);
```

The material is organized as follows: in Sect. 3.1 we introduce the back-end, that is to say, the portion of the implementation invisible to the user; in Sect. 3.2 we present the DSEL and explain how it transposes in back-end objects.

3.1. Algebraic back-end. In this section we introduce the foundation upon which rests the user-friendly interface. More specifically, in Sect. 3.1.1 we present vector spaces and discrete variables, which provide the basic representation of discrete functions in terms of vectors of DOFs. In Sect. 3.1.2 we propose an original implementation of linear operators with unconventional stencil. Its use in the context of FE-like matrix assembly is discussed in Sect. 3.1.3.

3.1.1. Vector spaces and discrete variables. We assume in what follows that a `Mesh` type is available defining (i) a positive integer `dim` corresponding to the space dimension; (ii) the subtypes `Cell`, `Face`, and `Node` for mesh elements of codimension 0, 1, and `dim` respectively. Mesh cells, faces and nodes inherit from the base class `Item` and are therefore collectively referred to as mesh items; (iii) the subtype `Pyramid` representing, for a given cell $T \in \mathcal{T}_h$ and a face $F \subset \mathcal{F}_T$, the pyramid $\mathcal{P}_{T,F}$. The relevant sets of mesh items can be accessed by the free functions listed in Table 3.1. The Lebesgue measure of a mesh item can be obtained via the free function `measure(const Item & I)`. Observe that the notion of geometric element type, standard in FE methods, is absent from the mesh concept, as it is irrelevant for the methods considered in this work.

Degrees of freedom are materialized by objects of type `DegreeOfFreedom`. Each instance of `DegreeOfFreedom` is attached to a mesh item which can be retrieved by the `DegreeOfFreedom` member function `item()`. The returned `Item` object can then be recast into the appropriate `Cell`, `Face`, or `Node` object in a safe way. The vector space of DOFs, denoted by $\mathbb{V}_h$ in Sect. 2.2, is represented by an instance of the class `VariableMng`, whereas the elements $\mathbf{v}_h \in \mathbb{V}_h$ are instances of the class `Variable`. The class `Variable` has two template parameters `ItemT` (either a `Cell`, `Face`, `Node`, or a `DegreeOfFreedom`) and `ValueT` (either `Real`, `RealVector`, or `RealMatrix`, hence handling scalar, vector and matrix valued variables) corresponds to a vector containing elements of type `ValueT` and indexed by `ItemT`s. Instances of the class `Variable` are managed by `VariableMng`, which handles memory in a centralized manner thus ensuring better performance. Access to variables is granted by `VariableMng` via a unique string key.

EXAMPLE 3.1 (Cell centered spaces of DOFs). *If* $\mathbb{V}_h = \mathbb{R}^{\mathcal{T}_h}$ *and* $\mathbf{v}_h \in \mathbb{V}_h$*, the notation* $\mathbf{v}_h = (v_T)_{T\in\mathcal{T}_h}$ *transposes in the random accessor* `vh[T]` *where* `vh` *is an instance of* `Variable<Cell,Real>` *and* `T` *is of type* `Cell`*.*

EXAMPLE 3.2 (Hybrid spaces of DOFs). *The hybrid space* $\mathbb{V}_h = \mathbb{R}^{\mathcal{T}_h} \times \mathbb{R}^{\mathcal{F}_h}$ *features degrees of freedom localized at both cells and faces. In this case, the implementation of Example 3.1 is modified as follows: (i) instances of* `DegreeOfFreedom` *are created for all cells in* `allCells(Th)` *and all faces in* `allFaces(Th)` *(cf. Table 3.1); (ii) vectors of DOFs are represented by instances of* `Variable<DegreeOfFreedom,ValueT>` *indexed by* `DegreesOfFreedom`*. Notice that the use of* `DegreeOfFreedom` *instead of* `Cell` *would also have been possible for Example 3.1. However, in our present implementation, the resulting code would have been slightly more inefficient.*

In what follows the reader can safely assume that all implementations of spaces of DOFs rely on `DegreeOfFreedom` as described in Example 3.2.

3.1.2. Linear operators with embedded stencil: The class `LinearCombination`. The point of view presented in Sect. 2 naturally leads to FE-like assembly of local contributions stemming from integrals over elements or faces. However, a few major differences have to be considered:

(i) the stencil of the gradient operator $\mathfrak{G}_h$ (and, consequently, of the piecewise affine reconstruction $\mathfrak{R}_h$) may vary from element to element, and is possibly data-dependent. This is the case, e.g., for the methods of Sect. 2.3 based on the L-construction;
(ii) additionally, the stencil may be non-local, since DOFs from neighboring elements may enter local reconstructions.

The above facts invalidate the classical approach based on a global table of DOFs inferred from a mesh and a finite element in the sense of Ciarlet [11, pag. 93]. Our solution to overcome the above difficulties while preserving the FE spirit of the assembly stage summarizes as follows:

(i) the notion of local element is dropped. Degrees of freedoms are globally indexed by mesh items (`Cell`, `Face`, or `Node`) or `DegreeOfFreedom`s as is the case for `Variable`s (cf. Sect. 3.1.1);
(ii) linear operators such as $\mathfrak{G}_h$, $\mathfrak{R}_h$, and trace reconstructions have embedded stencils.

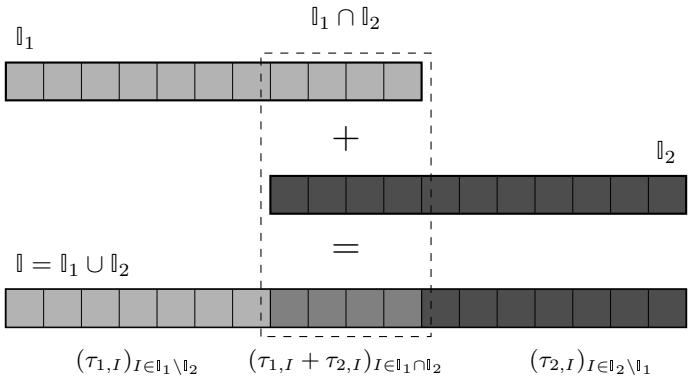


FIG. 3.1. *Computing the sum of two linear combinations* $\mathtt{l}_1 = (I, \tau_{1,I})_{I\in\mathbb{I}_1}$ *and* $\mathtt{l}_2 = (I, \tau_{2,I})_{I\in\mathbb{I}_2}$ *requires computing the intersection* $\mathbb{I}_1 \cap \mathbb{I}_2$ *and the union* $\mathbb{I}_1 \cup \mathbb{I}_2$

Linear operators with embedded stencil are represented by instances of the class `LinearCombination`, which realizes a linear application from $\mathbb{V}_h$ on the space of real tensors $\mathbb{T}_r$ of order r with r equal to 0 (scalar), 1 (vector), or 2 (matrix). `LinearCombination` has two templates parameters `ValueT` and `ItemT` representing, respectively, the value of the coefficients and the type of the item for indexing (`Cell`, `Face`, `Node`, or `DegreeOfFreedom`). The template parameter `ItemT` defaults to `DegreeOfFreedom` since this yields the more flexible (although not necessarily the most efficient) implementation as discussed in Example 3.2.

For a fixed $r \in \{0, 1, 2\}$, a `LinearCombination l` can be represented as a list of couples $(I, \tau_I)_{I\in\mathbb{I}}$ where $\mathbb{I} \subset \mathbb{V}_h$ is the stencil (implemented as a vector of global DOFs) and $(\tau_I)_{I\in\mathbb{I}}$, with $\tau_I \in \mathbb{T}_r$ for all $I \in \mathbb{I}$, is the corresponding vector of coefficients. The stencil $\mathbb{I}$ can be accessed by invoking

```
Vector<ItemT> LinearCombination<ValueT,ItemT>::stencil();
```

The evaluation at $\mathbf{v}_h \in \mathbb{V}_h$ is obtained by calling the function

```
ValueT
LinearCombination<ValueT,ItemT>::eval(Variable<ValueT,ItemT> & v_h)
{
  return \sum_{I\in\mathbb{I}} \tau_I v_I;
}
```

An efficient algebra for linear combinations is implemented by extending the operators defined for the type `ValueT`. Particular attention is required when dealing with expressions that contain the sum or subtraction of linear combinations, since this involves computing the intersection and the union of the corresponding stencils; see Figure 3.1.

Efficiency in this case is achieved by standard buffering techniques implemented via the auxiliary class `LinearCombinationBuffer` with template parameters `ValueT` and `ItemT` (defaulting to `DegreeOfFreedom`). Once all the algebraic operations are buffered, an object of type `LinearCombination` can be obtained calling the function

```
LinearCombination<ValueT,ItemT>
LinearCombinationBuffer<ValueT,ItemT>::linearCombination()
```

Unlike `LinearCombinationBuffer`, `LinearCombination` is optimized for fast traversing the vectors of stencils and coefficients, it supports efficient multiplication and division by constants, but it cannot be used in expressions involving the sum or subtraction of `LinearCombination`s. An example of usage is provided in Listing 2, where the implementation of the gradient reconstruction $\mathfrak{G}_h^{\mathrm{green}}$ defined by (2.12) is discussed. Observe that the gradient $\mathfrak{G}_h^{\mathrm{ccg}}$ can be obtained from the

same piece of code by simply changing the value returned by the trace reconstruction operator $\mathbf{T}_h$ in line 6.

Listing 2
Implementation of the gradient reconstruction $\mathfrak{G}_h^{\mathrm{green}}$ (2.12) *for an element* $T \in \mathcal{T}_h$.

```
1  // Cell centered DOF v_T as a scalar linear combination
2  LinearCombination<Real> vT(I_T, 1.);
3  // Algebraic operations are buffered to improve efficiency
4  LinearCombinationBuffer<RealVector> buffer;
5  for(F ∈ F_T) {
6    LinearCombination<Real> vF = T_h.eval(F);
7    buffer += (|F|/|T|)(vT-vF)n_{T,F};
8  }
9  // All buffered operations are actually performed here
10 LinearCombination<RealVector> GT = buffer.linearCombination();
```

A further refinement to improve performance is provided by the class `LinearCombinationMng` with template parameters `ValueT` and `ItemT` (defaulting to `DegreeOfFreedom`). This class can be used to perform operations on a set of linear combinations sharing the same values for the template parameters `ValueT` and `ItemT`. More specifically, `LinearCombinationMng` (i) allows to efficiently compute the intersection and union of linear combination stencils required to compute the sum of two linear combinations (see Figure 3.1); (ii) optimizes memory management by avoiding unnecessary copies while performing buffered operations.

3.1.3. Finite element-like assembly. `LinearCombination`s provide the key facilities to build local contributions stemming from integrals over elements or faces in the assembly step. To illustrate the main ideas, we start with an example.

Example 3.3 (Local left-hand side contribution). *For a given* $T \in \mathcal{T}_h$ *and for* u_h, $v_h \in V_h^{\mathrm{ccg}}$ *we consider the local contribution* $\mathbf{A}_{\mathrm{loc}}$ *associated to the term*

$$\int_T \kappa \nabla_h u_h \cdot \nabla_h v_h.$$

Both $(\kappa \nabla_h u_h)|_T = \kappa|_T \nabla(u_h|_T)$ *and* $(\nabla_h v_h)|_T = \nabla(v_h|_T)$ *are instances of* `LinearCombination`*, say* $\mathtt{l}_u = (\mathbb{J}, \tau_{u,J})_{J\in\mathbb{J}}$ *and* $\mathtt{l}_v = (\mathbb{I}, \tau_{v,I})_{I\in\mathbb{I}}$*. The local left-hand side contribution is*

$$\mathbf{A}_{\mathrm{loc}} = \left[|T| \tau_{v,I} \cdot \tau_{u,J}\right]_{I\in\mathbb{I},\, J\in\mathbb{J}}. \tag{3.1}$$

The stencils $\mathbb{I}$ *and* $\mathbb{J}$ *play the same role as the lines of the table of DOFs corresponding to test and trial functions supported in* T *in standard FE implementations. As such, they are related to the lines and columns of the global matrix* $\mathbf{A}$ *to which* $\mathbf{A}_{\mathrm{loc}}$ *contributes,*

$$\mathbf{A}(\mathbb{I}, \mathbb{J}) \leftarrow \mathbf{A}(\mathbb{I}, \mathbb{J}) + \mathbf{A}_{\mathrm{loc}}. \tag{3.2}$$

The programming counterpart of (3.2) *is given in Listing 3 (the function* `contract` *implements the tensor contraction corresponding to the rank of its arguments).*

Listing 3
Left-hand side assembly (`A` *represents here the global matrix)*

```
Cell T;
LinearCombination<RealVector> lu /* = ∇_h u_h */;
LinearCombination<RealVector> lv /* = ∇_h v_h */;
SparseRealMatrix A(/* ... */);
// Local contribution (see (3.1))
RealMatrix Aloc = contract(lv, measure(T)*K[T]*lu);
// Assembly (see (3.2))
assemble(A, lv.stencil(), lu.stencil(), Aloc);
```

REMARK 3.4 (Matrix assembly). *Coherently with the mathematical notation* (3.2)*, the assembly of the local contribution in line 8 of Listing 3 is realized by the function*

```
assemble(SparseRealMatrix & A,
         const Vector<DegreeOfFreedom> & I,
         const Vector<DegreeOfFreedom> & J,
         const RealMatrix & Aloc);
```

The actual indices of the lines and columns to which `Aloc` *contributes can be inferred from the arguments* `I` *and* `J` *(every* `DegreeOfFreedom` *stores the global number of the corresponding algebraic unknown).*

Generalizing Example 3.3, we notice that bilinear forms result from the sum of terms with the following general form:

$$\sum_{I \in \mathcal{I}_h} \int_I (\gamma_u \times \mathcal{L}_u(u_h)) \cdot (\gamma_v * \mathcal{L}_v(v_h)) \,, \tag{3.3}$$

where

(i) $\mathcal{I}_h \in \left\{\mathcal{T}_h, \mathcal{F}_h, \mathcal{F}_h^i, \mathcal{F}_h^b\right\}$ is a set of mesh items (cf. Sect. 3.1.1);
(ii) γ_u and γ_v are tensor fields of rank r_{γ_u} and r_{γ_v} respectively possibly depending on constants and on discrete variables;
(iii) $\mathcal{L}_u$ is a linear operator acting on the trial function $u_h \in U_h$ and yielding a tensors-valued field of order r_u. The operator $\mathcal{L}_u$ is represented by an instance of `LinearCombination`;
(iv) $\mathcal{L}_v$ is a linear operator acting on the test function $v_h \in V_h$ (which can possibly belong to a space $V_h \neq U_h$) and yielding a tensor-valued field of order r_v. The operator $\mathcal{L}_v$ is represented by an instance of `LinearCombination`;
(v) $\times$ (resp. $*$) is an admissible product between a tensor of order r_{γ_u} (resp. r_{γ_v}) and a tensor of order r_u (resp. r_v) yielding a tensor-valued field with order r;
(vi) $\cdot$ is the contraction product for tensors of order r.

The factors $(\gamma_u \times \mathcal{L}_u(u_h))$ and $(\gamma_v * \mathcal{L}_v(v_h))$ are respectively referred to as *trial* and *test expressions*.

EXAMPLE 3.5 (Bilinear term). *The term considered in Example 3.3 can be recast into the form* (3.3) *by setting* $\mathcal{I}_h = \mathcal{T}_h$*,* $\gamma_u = \kappa$ *and* $r_{\gamma_u} = 2$*,* $\gamma_v = 1$ *and* $r_{\gamma_v} = 0$*,* $\mathcal{L}_u = \nabla_h$ *and* $r_u = 1$*,* $\mathcal{L}_v = \nabla_h$ *and* $r_v = 1$*, and denoting by* $\times$*,* $*$*, and* $\cdot$ *the standard matrix-vector product, the scalar product, and the standard vector inner product respectively.*

Right-hand side contributions can be handled in a similar fashion. The details are omitted for brevity.

3.2. Functional front-end. In this section we discuss the front-end of the language. More specifically, in Sect. 3.2.1 we introduce function spaces, which provide a functional representation for discrete functions building upon vector spaces. In Sect. 3.2.2 we address linear and bilinear expressions, which are the main building blocks for the programming counterpart of linear and bilinear forms. For the sake of clarity, the discussions in both Sections 3.2.1 and 3.2.2 are focused on the scalar case. The extension to the vector case is briefly addressed in Sect. 3.2.3. By the end of this section, all the elements of Listing 1 should become familiar to the reader.

3.2.1. Function spaces. The programming counterparts of incomplete broken polynomial spaces defined by (2.7) are classes that match the concept defined in Listing 4.

LISTING 4
Function space concept

```
struct FunctionSpaceConcept {
  class DiscreteFunction; // See Listing 5 for details
  class TrialFunction;    // See Listing 6 for details
  class TestFunction;     // See Listing 7 for details
  // Constructor
  FunctionSpaceConcept(const Mesh & Th);
  // Create a general discrete function
```

```
  DiscreteFunction discreteFunction(const String & key) const;
  // Create a trial function
  TrialFunction trialFunction(const String & key) const;
  // Create a test function
  TestFunction testFunction(const String & key) const;
  // Constant value of 𝔊_h|_S for S ∈ 𝒮_h expressed as a linear combination of DOFs
  LinearCombination<RealVector> Gh(S) const;
  // Value of ℜ_h|_S(x) for x ∈ S and S ∈ 𝒮_h expressed as a linear combination of
DOFs
  LinearCombination<Real> Rh(S, x) const;
};
```

The key role of a function space is to bridge the gap between the algebraic representation of DOFs and the functional representation used in the methods of Sect. 2. This is achieved by the functions `FunctionSpaceConcept::Gh` and `FunctionSpaceConcept::Rh`, which are the `C++` counterparts of the linear operators $\mathfrak{G}_h$ and $\mathfrak{R}_h$ respectively; see Sect. 2.1. More specifically,

(i) for all $S \in \mathcal{S}_h$, `FunctionSpaceConcept::Gh(`S`)` returns a vector-valued linear combination corresponding to the (constant) restriction $\mathfrak{G}_h|_S$;

(ii) for all $S \in \mathcal{S}_h$ and all $\mathbf{x} \in S$, `FunctionSpaceConcept::Rh(`S`, x)` returns a scalar-valued linear combination corresponding to $\mathfrak{R}_h|_S(\mathbf{x})$ defined according to (2.6).

The `LinearCombination`s returned by `Gh` and `Rh` can be used to build linear and bilinear contributions as described in Sect. 3.1.3.

A function space also defines the functional subtypes `DiscreteFunction`, `TestFunction` and `TrialFunction` corresponding to the mathematical notions of discrete functions, test and trial functions respectively. `TrialFunction`s and `DiscreteFunction`s are associated to a `Variable` object containing the corresponding vector of DOFs. For `DiscreteFunction`s, the vector of DOFs is used for the evaluation at a point $\mathbf{x} \in S$, $S \in \mathcal{S}_h$; cf. Listing 5. `TestFunctions` are used to represent the basis of the discrete space, and are not associated to a vector of DOFs. As to `TrialFunction`s, the implementation is similar to `TestFunctions` and it has additionally access to the solution of the discrete problem.

Unlike `DiscreteFunction`s, `TrialFunction`s and `TestFunction`s have partially lazy evaluation mechanisms, i.e., the evaluation returns a `LinearCombination` instead of a value; cf. Listings 6 and 7. The main motivation for introducing functional subtypes is to avoid differentiating DSEL keywords for test and trial functions as is currently the case in `Feel++`; cf. Sect. 3.2.2.

Listing 5
Mockup of discrete function

```
struct DiscreteFunction {
  inline Real eval(S, x) const {
    return m_Vh->Rh(S, x).eval(m_DOFs);
  }
  inline RealVector grad(S) const {
    return m_Vh->Gh(S).eval(m_DOFs);
  }
private:
  MyFunctionSpace * m_Vh;
  Variable<Real> m_DOFs;
};
```

Listing 6
Mockup of trial function

```
struct TrialFunction {
  inline LinearCombination<Real> eval(S, x) const {
    return m_Vh->Rh(S, x);
  }
```

```
  inline LinearCombination<RealVector> grad(S) const {
    return m_Vh->Gh(S);
  }
private:
  MyFunctionSpace * m_Vh;
  Variable<Real> m_DOFs;
};
```

Listing 7
Mockup of test function

```
struct TestFunction {
  inline LinearCombination<Real> eval(S, x) const {
    return m_Vh->Rh(S, x);
  }
  inline LinearCombination<RealVector> grad(S) const {
    return m_Vh->Gh(S);
  }
private:
  MyFunctionSpace * m_Vh;
};
```

Actual function spaces can be generated via a helper template class `FunctionSpace` with two template parameters (labels are defined using the `boost::parameter` library):

(i) a template parameter identified by the label **poly** (equal to **poly<0>** or **poly<1>**) and corresponding to the smallest containing polynomial space (either $\mathbb{P}_d^0(\mathcal{T}_h)$ or $\mathbb{P}_d^1(\mathcal{T}_h)$);
(ii) a template paramater identified by the label **gradient** and corresponding to a (piecewise constant) gradient reconstruction.

The gradient reconstruction fixes both the vector space of DOFs $\mathbb{V}_h$ according to (2.7) and the choice (2.3) for $\mathcal{S}_h$. Gradient reconstructions are generated via the helper template class `GradientReconstruction` from the following template arguments:

(i) a template parameter identified by the label **submesh** (equal to `Pyramidal` or `Identity`). This corresponds to the choice (2.3) for the auxiliary submesh;
(ii) a template parameter identified by the label `dof` (equal to `CellCentered` or `Hybrid`) corresponding to the choice (2.5);
(iii) a template parameter identified by the label `interpolator` which corresponds to a trace interpolator such as the one defined by (2.11). The default value for this parameter is `NoInterpolator` meaning that face unknowns are not interpolated.

It is important to observe that the template parameter `dof` is not redundant since the space of DOFs cannot be, in general, deduced from the `interpolator`. Indeed, following [24], it is possible to conceive a trace interpolator that automatically decides whether to keep a face unknown or to interpolate it according to the problem data. In this case, while `dof` should be set equal to `Hybrid`, the actual size of the space of DOFs will only be determined at run time. The programming counterparts of the gradient reconstructions and discrete spaces discussed in Sect. 2 are listed in Tables 3.2 and 3.3 respectively.

Example 3.6 (Matrix assembly for the SUSHI method). *Using function spaces, test and trial functions the assembly of the matrix corresponding to the bilinear form a_h^{sushi} defined by* (2.19) *can be obtained as described in Listing 8.*

Listing 8
Matrix assembly for the SUSHI method

```
Mesh Th(/* ... */);
Variable<Cell,RealMatrix> K; // Diffusion tensor κ
FunctionSpace<Mesh,
              span< poly<1>, gradient<SUSHIHGradient>  >
```

```
              >::type Vh(Th);
auto uh = Vh.trialFunction();
auto vh = Vh.testFunction();
SparseRealMatrix A(/* ... */);
for( auto T: allCells(Th)) {
  for( auto S : allPyramids(T)) {
  // retrieve the LinearCombination associated to the gradient
  // of trial and test functions
  auto Gu = uh.grad(S);
  auto Gv = vh.grad(S);
  assemble(A,
           Gv.stencil(),
           Gu.stencil(),
           contract(Gv, measure(S)*K[T]*Gu)
           );
  }
}
```

3.2.2. Linear and bilinear expressions. The main goal of the DSEL is to allow a notation as close as possible to that of the mathematical counterpart described in Sect. 2. The focus of this section is on bilinear forms, as the ingredients for linear forms are essentially similar. In what follows we exemplify production rules for trial and test expressions as well as bilinear terms of the form (3.3) using the Extended Backus–Naur Form (EBNF), see [1]. The exposition is not meant to be exhaustive, but instead to present a few significant examples from which others can be inferred. The actual implementation is based on the `boost::proto` library by Niebler [32].

Terminals. The terminals of the DSEL is composed of a number predefined types categorized in the following families:

- the `BaseType` family for the standard `C++` types representing integers and reals;
- the `VarType` family for all discrete variable types defined in Sect. 3.1.1;
- the `MeshGroupType` family for types representing collection of mesh entities such as the ones listed in Table 3.1;
- the `DiscreteFunction`, `TestFunction` and `TrialFunction` families representing the discrete functions, test and trial functions defined in Sect. 3.2.

Trial and test expressions. Trial (resp. test) expressions are obtained as the product of a coefficient γ_u (resp. γ_v) by a linear operator $\mathcal{L}_u$ (resp. $\mathcal{L}_v$) acting on a trial (resp. test) function. The coefficient can result from the algebraic combination of constant values and `Variable`s evaluated at item I (cf.(3.3)). Listing 9 displays a few production rules for coefficients involving, in particular, constant values, `Variable`s over `Cell`s and products thereof.

Listing 9
Examples of production rules for the coefficient γ in (3.3)

```
BaseExpr = BaseType | BaseExpr * BaseExpr;

VarExpr  = VarType | BaseExpr * VarExpr | VarExpr * VarExpr;

CoefExpr = BaseExpr | VarExpr;
```

To obtain trial and test expressions, we introduce linear operators acting on test and trial functions. A few examples are provided in Listing 10, and include (i) **id**, the value of the trial/test function; (ii) **grad**, the gradient of the trial/test function; (iii) trace operators like **jump** and **avg** representing, respectively, the jump and average of a trial/test function across a face. Besides linear operators, the production rules for trial and test expressions in Listing 10 include various products by coefficients resulting from the production rules of Listing 9 (**dot** and **ddot** denote, respectively, the vector inner product and the contraction for tensors of order 2).

Listing 10
Production rules for trial and test expressions

```
LinearOperator = "id" | "grad" | "jump" | "avg";

TrialExpr = TrialFunction                  |
            CoefExpr * TrialExpr           |
            "dot("CoefExpr, TrialExpr")"   |
            "ddot("CoefExpr, TrialExpr")"  |
            LinearOperator"("TrialExpr")";

TrialExpr = TestFunction                  |
            CoefExpr * TestExpr           |
            "dot("CoefExpr, TestExpr")"   |
            "ddot("CoefExpr, TestExpr")"  |
            LinearOperator"("TestExpr")";
```

Bilinear forms. Once test and trial expressions are available, bilinear terms can be obtained as contraction products of trial and test expressions or as sums, as described in Listing 11.

Listing 11
Production rules for bilinear terms

```
BilinearTerm = TrialExpr * TestExpr          |
               "dot("TrialExpr, TestExpr")"  |
               "ddot("TrialExpr, TestExpr")" |
               CoefExpr * BilinearTerm       |
               BilinearTerm + BilinearTerm;
```

Bilinear forms finally result from the integration of bilinear terms on groups of mesh items (cf. Table 3.1). Production rules for bilinear forms are given in Listing 12. Observe that **integrate** acts as a binary operator that takes as arguments the group of items over which integration is performed and the bilinear term to integrate. Compile-time checks are performed to ensure that the expression corresponding to the bilinear term is meaningful with respect to the type of items we are integrating on. For example, expressions containing the operators **jump** and **avg** will be rejected when integrating over cells.

Listing 12
Production rules for bilinear forms

```
IntegrateBilinearTerm = "integrate("MeshGroup, BilinearTerm")";
BilinearForm = IntegrateBilinearTerm                  |
               IntegrateBilinearTerm + BilinearForm;
```

In the `boost::proto` based implementation, the production rules of Listings 9, 10, 11, and 12 yield expression trees where each node is an object of type `expr` identified by a tag. The leafs of the tree are occupied by terminal expressions including base types and variables (cf. Listing 9), meshes (cf. Listing 12), test and trial functions (cf. Listing 10).

Bilinear expressions are used to define objects of type `BilinearForm`. These objects store the bilinear expression in a collection of generated structures of base type `IBaseTerm` that wrap bilinear terms. A standard mechanism based on the visitor pattern allows to evaluate each bilinear term using its true type.

Example 3.7 (Bilinear form for the SUSHI method). *The programming counterpart of the bilinear form* a_h^{sushi} *defined by* (2.19) *is given in Listing 13. The corresponding expression tree is detailed in Fig. 3.2.*

Listing 13
DSEL based implementation of the bilinear form a_h^{sushi} *defined by* (2.19)

```
1 Mesh Th(/* ... */);
```

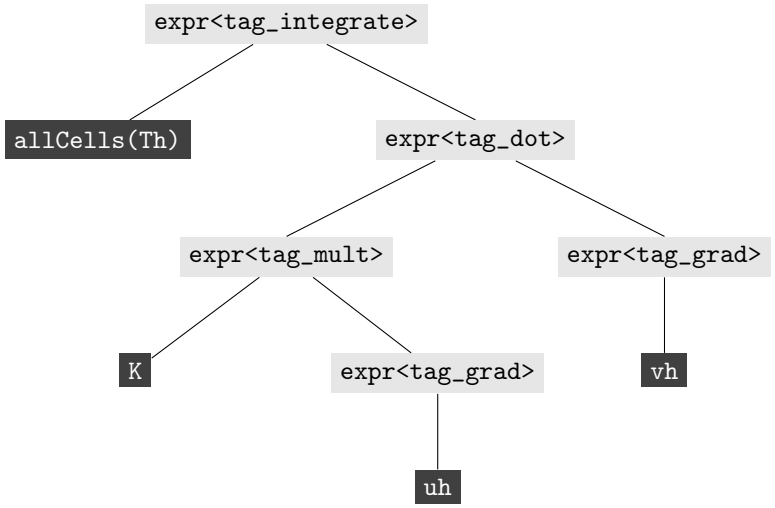


FIG. 3.2. *Expression tree for the bilinear form defined at line 10 of Listing 13. Expressions are in* light gray*, language terminals in* dark gray

```
2  FunctionSpace<Mesh,
3                span< poly<1>, gradient<SUSHIHGradient> >
4               >::type Vh(Th);
5  auto uh = Vh.trialFunction();
6  auto vh = Vh.testFunction();
7  // Observe that the language automatically handles the fact that gradients are piecewise
   constant
8  // over pyramids rather than cells
9  BilinearForm ah = integrate(allCells(Th),
10                             dot(K*grad(uh),grad(vh));
```

The language is completed by specific keywords such as, e.g., **N()** and **H()**, which, for all $F \in \mathcal{F}_h$, return the precomputed values of n_F and h_F.

Context-based evaluation. Following the philosophy of `Feel++` [33] and `boost::proto`, the evaluation of linear and bilinear expressions is context-dependent. Contexts are objects that prescribe an evaluation policy for each node type. More precisely, when an expression is evaluated, the context is invoked at each node of the tree. When evaluating bilinear expressions, the context provides a representation for the global matrix that matches the assumption discussed in Remark 3.4. The integration of a generic bilinear expression is then obtained as described in Listing 14. The function `bln_integrate` is invoked when evaluating nodes of type `expr<tag_integrate>` such as the root of the tree in Figure 3.2. Observe that switching to a matrix-free version only requires to define an appropriate evaluation context (which is initialized with the vector by which the matrix is right-multiplied).

LISTING 14

Assembly of the generic bilinear contribution (3.3). *The function* `bln_integrate` *generalizes the example in Listing 8*

```
1  template<typename ItemT, typename BilinearExprT, typename ContextT>
2  void bln_integrate(const GroupT<ItemT> & group,
3                     const BilinearExprT & expr,
4                     const ContextT & ctx)
5  {
6    // Retrieve trial and test expressions
7    auto trial_expr = getTrialExpr(expr);
8    auto test_expr  = getTestExpr(expr);
9    // Retrieve matrix.  Matrix representation is context-dependent
```

```
10    auto A = ctx.getMatrix();
11    for(auto item : group) { // ∑_{I∈I_h} ...
12      auto Lu = eval(trial_expr, item);   // (γ_u × L_u(u_h))|_I
13      auto Lv = eval(test_expr, item);    //(γ_v * L_v(v_h))|_I
14      // Assemble local contribution
15      assemble(A,
16               Lu.stencil(),                 // line indices
17               Lv.stencil(),                 // column indices
18               contract(Lv, measure(I)*Lu) // |I|(γ_u × L_u(u_h))|_I · (γ_v * L_v(v_h))|_I
19               );
20    }
21  }
```

A few comments on Listing 14 are required. Lines 7 and 8 make use of the functions `getTestExpr` and `getTrialExpr` to retrieve the test and trial components of the bilinear expression. These functions are implemented essentially in the same spirit as the `boost::proto` functions `left` and `right`. The evaluation of trial and test expressions at lines 12 and 13 returns instances of `LinearCombination` (with suitable instances for the template parameters), which are available for use in the local assembly at line 16.

3.2.3. The vector case. In this section we briefly address the extensions of the grammar to handle vector problems. This allows to write the programming counterpart of the method (2.26) as detailed in Listing 15.

Listing 15
Implementation of the method (2.26) *for the Stokes problem*

```
1  Mesh Th(/* ... */);
2  Real eta;
3  // Function spaces
4  FunctionSpace<Mesh,
5                span< poly<1>, gradient<GreenGradient> >
6                >::type Uh(Th);
7  FunctionSpace<Mesh,
8                span< poly<0>, gradient<NullGradient> >
9               >::type Ph(Th);
10 // Test and trial functions
11 auto uh = Uh.trialVectorFunction("uh");
12 auto vh = Uh.testVectorFunction("vh");
13 auto ph = Ph.trialFunction("ph");
14 auto qh = Ph.testFunction("qh");
15 // Penalty parameter
16 auto penalty = eta/H();
17 // Indices
18 Range range(dim);
19 auto _i = range.get();
20 // Diffusive term (2.23)
21 BilinearForm ah =
22   integrate(allCells(Th),
23             sum(_i)( dot(grad(uh(_i)),grad(vh(_i))))
24            )
25 + integrate(allFaces(Th),
26             sum(_i)( - dot(N(),avg(grad(uh(_i))))*jump(vh(_i))
27                      - jump(uh(_i))*dot(N(),avg(grad(vh(_i))))
28                      + penalty*jump(uh(_i))*jump(vh(_i))
```

```
29                      )
30              ) ;
31  // Pressure gradient (2.24)
32  BilinearForm bh = integrate(allCells(Th), -ph*div(vh))
33                  + integrate(allFaces(Th),
34                              avg(ph)*dot(N(),jump(vh))
35                              );
36  // Velocity divergence (2.24)
37  BilinearForm bth = integrate(allCells(Th), div(uh)*qh)
38                   + integrate(allFaces(Th),
39                               -dot(N(),jump(uh)) * avg(qh)
40                               );
41  // Inf-sup stabilization (2.25)
42  BilinearForm sh = integrate(internalFaces(Th),
43                              H()*jump(ph)*jump(qh)
44                              );
```

The following extensions are introduced to handle the vector case:

(i) the concept of expression rank allows to classify the expression as `Scalar`, `Vector`, or `Tensor`;

(ii) the classes `VectorFunction`, `TestVectorFunction`, and `TrialVectorFunction` are introduced to represent discrete vector functions. An example is provided in lines 11 and 12 of Listing 15;

(iii) the components of discrete vector functions can be traversed using the `Range` concept; cf. lines 18 and 19 of Listing 15.

Vector and tensor terminals dispose of an **operator**`()(/*index */)` operator that returns an object of type `ScalarView<ExprT>` to access the expression component specified by an index iterator. The unary operator `sum(/*range */)` returns an expression node which allows to iterate over its child nodes corresponding to the components of the vector expression. The bilinear forms `ah`, `bh`, `bth`, and `sh` defined in Listing 15 can either contribute to a unique global matrix in the context of monolithic methods, or be used to assemble the corresponding submatrices when, e.g., pressure correction methods are used to march in time.

4. Numerical examples. In this section we provide a few numerical examples to assess the performance of the DSEL. Although the codes are

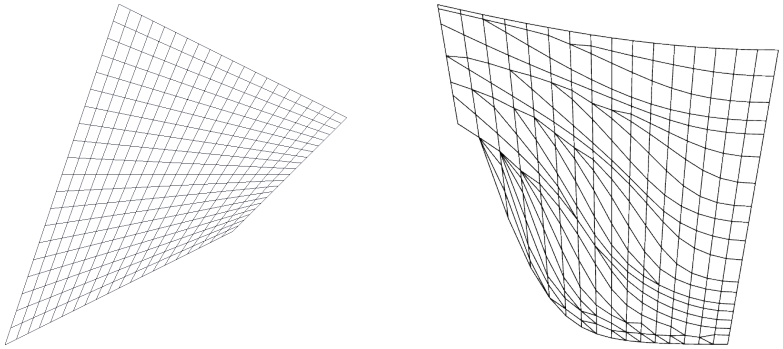
4.1. Codes description. The performance of the DSEL-based implementation of lowest-order methods discussed in Sect. 3 is compared with

- `Feel++`, an open source FE library whose main developer is one of the authors [35]. When possible, `Feel++` is used for comparison with more standard FE methods both in terms of accuracy and performance. The DSEL implemented in `Feel++` has profoundly inspired the present work;
- `fvC++`, an `stl`-based implementation of the back-end discussed in Sect. 3.1 developed by one of the authors and used in [13–15]. The matrix assembly in `fvC++` closely resembles Listing 8. No language facility is offered in this case.

The three codes are compiled with the `gcc` 4.5 compiler with the following compile options:

```
-03 -fno-builtin
-mfpmath=sse -msse -msse2 -msse3 -mssse3 -msse4.1 -msse4.2
-fno-check-new -g -Wall -std=c++0x
--param -max-inline-recursive-depth=32
--param max-inline-insns-single=2000
```

The benchmark test cases are run on a work station with a quad-core Intel Xeon processor GenuineIntel W3530, 2.80GHz, 8MB for cach size.



(a) Skewed quadrangular mesh

(b) Stratigraphic mesh. The actual aspect ratio is 10:1

FIG. 4.1. *Elements of the mesh families used in the benchmark*

4.1.1. Meshes. In our numerical tests we consider the following families of h-refined meshes: (i) the skewed quadrangular mesh family of Figure 4.1(a) generated using Gmsh [26] is used for the benchmarks of Sections 4.3 and 4.4; (ii) the stratigraphic mesh family of Figure 4.1(b) representing a geological basin is used for the benchmark of Sect. 4.5. This mesh family mixes triangular and quadrangular elements. The actual aspect ratio of the mesh is 10:1, resulting in elongated elements and sharp angles.

4.1.2. Solvers. The linear systems are solved using the PETSc library. For the diffusion benchmark of Sect. 4.3, we use the BICGStab solver preconditioned by the euclid ILU(2) preconditioner, with relative tolerance set to 10^{-13}. For the Stokes benchmark of Sect. 4.4, we use the GMRes solver with a ILU(3) preconditioner and a relative tolerance of 10^{-13}. The constant null space constraint option is activated to solve the system, and the resulting discrete pressure is scaled to ensure that the zero-mean constraint (2.21d) is satisfied. Note that our objective is not to test the solvers but rather compare for a given solution strategy the behavior of the various methods exposed in Sect. 2 as well as more conventional FE methods.

4.2. Benchmarks metrics. The benchmarks proposed in this section monitor various metrics:

(i) *Accuracy.* The accuracy of the methods is evaluated in terms of the L^2- and of discrete energy-norms of the error. For the methods of Sect. 2, the L^2-norm of the error is evaluated using the cell center as a quarature node, i.e.,

$$\|u - u_h\|_{L^2(\Omega)} \approx \left(\sum_{T \in \mathcal{T}_h} |T| (u(\mathbf{x}_T) - u_T)^2 \right)^{\frac{1}{2}}.$$

The actual definition of the discrete energy-norm is both problem- and method-dependent. Further details are provided for each test case. The convergence order of a method is classically expressed relating the error to the meshsize h.

(ii) *Memory consumption.* When comparing methods featuring different number of unknowns and stencils, a more fair comparison in terms of system size and memory consumption is obtained relating the error to the number of DOFs (N_{DOF}) and to the number of nonzero entries of the corresponding linear system (N_{nz}).

(iii) *Performance.* The last set of parameters is meant to evaluate the CPU cost for each method and implementation. To provide a detailed picture of the different stages and estimate the overhead associated to the DSEL, we separately evaluate

- t_{init}, the time to build the discrete space;
- t_{ass}, the time to fill the linear systems (local/global assembly). When DSEL-based implementations are considered, this stage carries the additional cost of evaluating the

expression tree for bilinear and linear forms;
- t_{solve}, the time to solve the linear system.

An important remark is that, in the context of nonlinear problems on fixed meshes, t_{init} often corresponds to precomputation stages, while t_{ass} contributes to each iteration.

4.3. Benchmark: Pure diffusion. Our first benchmark is based on the following exact solution for the diffusion problem (2.3):

$$u(\mathbf{x}) = \sin(\pi x_1)\cos(\pi x_2), \qquad \kappa = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

with suitable right-hand side f and Dirichlet boundary condition on $\partial\Omega$. The problem is solved on the skewed mesh family depicted in Figure 4.1(a). We compare the following methods: (i) the DSEL and `fvC++` implementations of the ccG method (2.15). The DSEL implementation is provided in Listings 1; (ii) the DSEL implementation of the SUSHI method with face unknowns (2.18) provided in Listing 13; (iii) the `Feel++` implementation of the first-order Rannacher–Turek elements $\mathbb{R}\square\mathbb{T}\mathbb{u}^1$; (iv) the `Feel++` implementation of $\mathbb{Q}^k$ elements with $k \in \{1, 2\}$. Since the solution is smooth, the $\mathbb{Q}^2$ element is expected to yield better performance. In real-life applications, however, the regularity of the solution is limited by the heterogeneity of the diffusion coefficient; see [16] and references therein for a discussion.

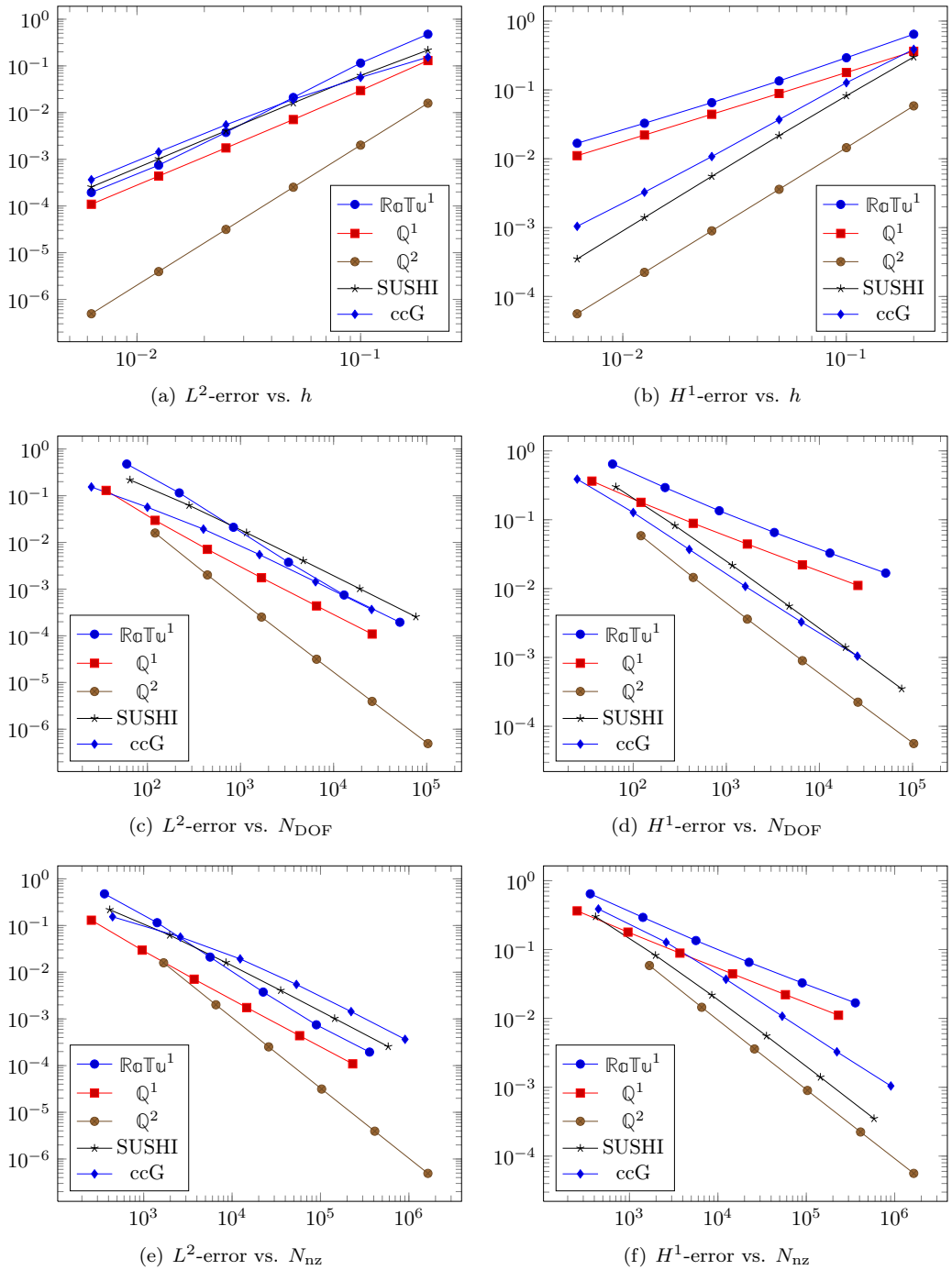
The accuracy and memory consumption analysis is provided in Figure 4.2. The discrete H^1-norm coincides with the natural coercivity norm for the method; see [15, 24] for further details on the SUSHI and ccG methods. As expected, the higher-order method $\mathbb{Q}^2$ elements yields better performance irrespectively whether the error is related to the meshsize h, the number of DOFs N_{DOF}, or the number of nonzero elements in the matrix N_{nz}. It has to be noted, however, that both the SUSHI and the ccG methods exhibit superconvergence in the discrete H^1-norm, thereby providing a better approximation of the gradient with respect to the first-order element methods.

The CPU cost analysis is provided in Figures 4.3 and 4.4. The cost of each stage of the computation is related to the number of DOFs in Figure 4.3 to check that the expected complexity is achieved. This is the case for all the methods considered. A comparison in terms of absolute computation time is provided in Figure 4.4. Overall, the initialization and assembly steps appear more expensive for the lowest-order methods. The overhead of the DSEL can be estimated by comparing with the `fvC++` implementation of the ccG. Some remarks are of order. (i) the main interest of the lowest-order methods presented in Sect. 2 is that general meshes can be handled seamlessly. For an example based on a less conventional mesh see Sect. 4.5. When a classical FE implementation is possible, the approach based on a reference element and a table of DOFs can be expected to overperform the `LinearCombination`-based handling of degrees of freedom; (ii) the FE code `Feel++` is a more mature project, which benefits from some degree of optimization and finer tuning; (iii) even when the overhead of the DSEL is disregarded, the `stl`-based implementation of `LinearCombination` in `fvC++` yields similar performance as the dedicated implementation used in the DSEL version.

4.4. Benchmark: Stokes. We consider the following analytical solution of the Stokes problem (2.21):

$$u_1(\mathbf{x}) = -\exp(x_1)(x_2\cos(x_2)+\sin(x_2)), \quad u_2(\mathbf{x}) = \exp(x)x_2\sin(x_2), \quad p(\mathbf{x}) = 2\exp(x_1)\sin(x_2)-\overline{p},$$

where $\overline{p}$ is chosen in such a way that the constraint (2.21d) is verified. The problem is solved on the skewed mesh family depicted in Figure 4.1(a). We compare the following methods: (i) the ccG method (2.26); (ii) an inf-sup stable method based on first-order Rannacher–Turek $\mathbb{R}\square\mathbb{T}\mathbb{u}^1$ elements for the velocity and $\mathbb{Q}^0$ elements for the pressure; (iii) an inf-sup stable method based on second-order $\mathbb{Q}^2$ elements for the velocity and $\mathbb{Q}^1$ elements for the pressure. The error is measured in terms of the L^2-norm for both the velocity and the pressure. The energy-norm of the error is



(a) L^2-error vs. h (b) H^1-error vs. h

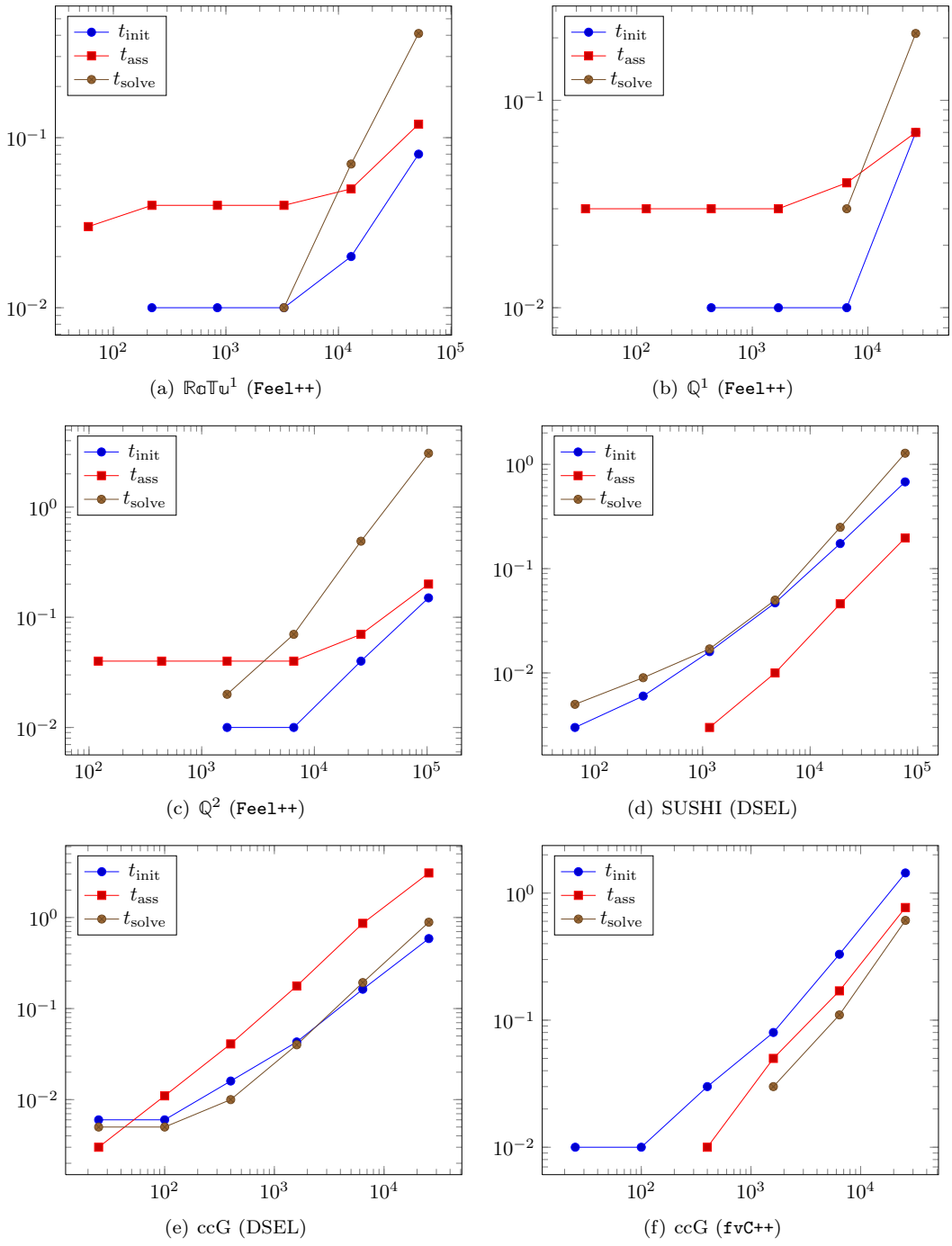
(c) L^2-error vs. N_{DOF} (d) H^1-error vs. N_{DOF}

(e) L^2-error vs. N_{nz} (f) H^1-error vs. N_{nz}

FIG. 4.2. *Accuracy and memory consumption analysis for the example of Sect. 4.3*

defined as follows:

$$\mathcal{E}_{\mathrm{sto}}(u_h,p_h)^2 := \|\nabla u - \nabla_h u_h\|^2_{L^2(\Omega)^{d,d}} + \|p-p_h\|^2_{L^2(\Omega)} + \alpha\left(\sum_{F\in\mathcal{F}_h} h_F^{-1}\|[\![u_h]\!]\|^2_{L^2(F)^d} + \sum_{F\in\mathcal{F}_h^i} h_F\|[\![p_h]\!]\|^2_{L^2(F)}\right),$$



(a) $\mathbb{R}\mathbb{T}^1$ (`Feel++`) (b) $\mathbb{Q}^1$ (`Feel++`)

(c) $\mathbb{Q}^2$ (`Feel++`) (d) SUSHI (DSEL)

(e) ccG (DSEL) (f) ccG (`fvC++`)

FIG. 4.3. *Performance analysis for the example of Sect. 4.3*

where $\alpha = 1$ for the ccG method and $\alpha = 0$ for both the $\mathbb{R}\mathbb{T}^1 - \mathbb{Q}^0$ and the $\mathbb{Q}^2 - \mathbb{Q}^1$ methods.

The accuracy and memory consumption analysis for the Stokes benchmark is provided in Figure 4.5. As expected, the higher-order method benefits from the regularity of the solution and is therefore more efficient. The results in terms of the L^2-error on the velocity are comparable for both the ccG and the $\mathbb{R}\mathbb{T}^1 - \mathbb{Q}^0$ methods, whereas the ccG method has a slight edge when it comes to the L^2-norm of the pressure. As regards the energy norm, the differences between the $\mathbb{R}\mathbb{T}^1 - \mathbb{Q}^0$ and the ccG methods are essentially related to the presence of the jumps of the

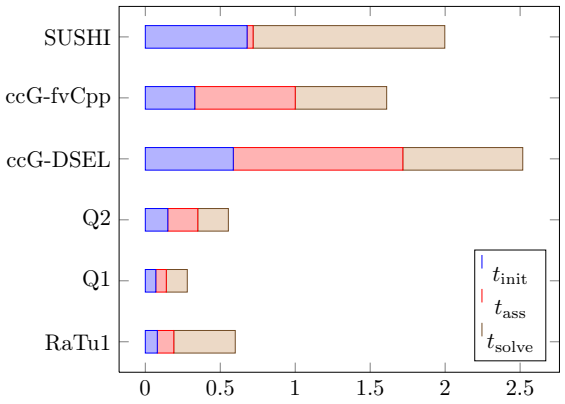


FIG. 4.4. *Comparison of different methods and implementation for the test case of Sect. 4.3 (time vs.* N_{DOF}*,* $h = 0.00625$*)*

pressure in the energy norm for the latter. An interesting remark is that the superconvergence phenomenon observed in the example of Sect. 4.3 for the ccG method is no longer present here.

The performance analysis for the Stokes benchmark is provided in Figure 4.5. Similar considerations hold as for the benchmark of Sect. 4.3. In this case, however, t_{solve} largely dominates $t_{\text{init}} + t_{\text{ass}}$ (especially for the $\mathbb{Q}^2 - \mathbb{Q}^1$ method), and it is excluded from the overall time comparison in Figure 4.7 to improve readability.

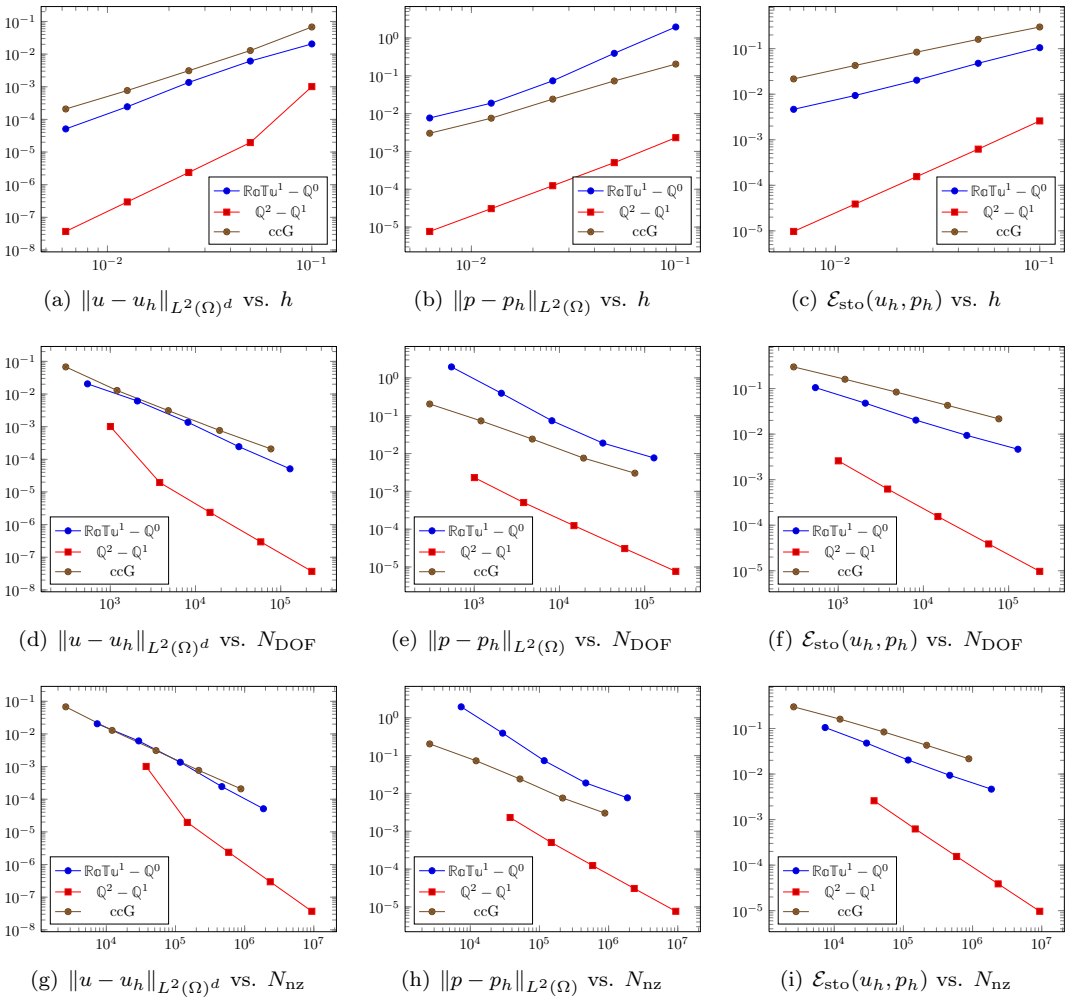
4.5. A problem in basin modeling. The last problem is based on the basin mesh family depicted in Figure 4.1(b) which contains both triangular and quadrangular elements. Handling this kind of meshes usually requires some specific modifications in finite element codes, since two reference finite elements exists. A key advantage of the lowest-order methods considered in the present work is that their construction remains unchanged for elements of different shape. We consider the anisotropic test case of [4],

$$u(\mathbf{x}) = \sin(\pi x_1)\sin(\pi x_2), \qquad \kappa = \begin{bmatrix} \epsilon & 0 \\ 0 & 1 \end{bmatrix},$$

with suitable right-hand side f and Dirichlet boundary conditions on $\partial\Omega$. The anisotropy ratio ϵ is taken equal to 0.1. We compare the following discretizations: (i) the G-method (2.10) whose DSEL implementation is provided Listing 16; (ii) the ccG method (2.15); (iii) the SUSHI method (2.18) with discrete gradient (2.20) expressed in terms of cell unknowns only.

LISTING 16
Implementation of the G-method (2.10)

```
Mesh Th(/* ... */);
FunctionSpace<Mesh,
              span< poly<0>, gradient<NullGradient> >
             >::type Vh(Th);
FunctionSpace<Mesh,
              span< poly<1>, gradient<GGradient> >
             >::type Uh(Th);
auto uh = Uh.trialFunction("uh");
auto vh = Vh.testFunction("vh");
BilinearForm ah = integrate(allFaces(Th),
                            -dot(N(),avg(K*grad(uh)))*jump(vh)
                            );
LinearForm bh = integrate(allCells(Th), f*v);
```



(a) $\|u - u_h\|_{L^2(\Omega)^d}$ vs. h (b) $\|p - p_h\|_{L^2(\Omega)}$ vs. h (c) $\mathcal{E}_{\mathrm{sto}}(u_h, p_h)$ vs. h

(d) $\|u - u_h\|_{L^2(\Omega)^d}$ vs. N_{DOF} (e) $\|p - p_h\|_{L^2(\Omega)}$ vs. N_{DOF} (f) $\mathcal{E}_{\mathrm{sto}}(u_h, p_h)$ vs. N_{DOF}

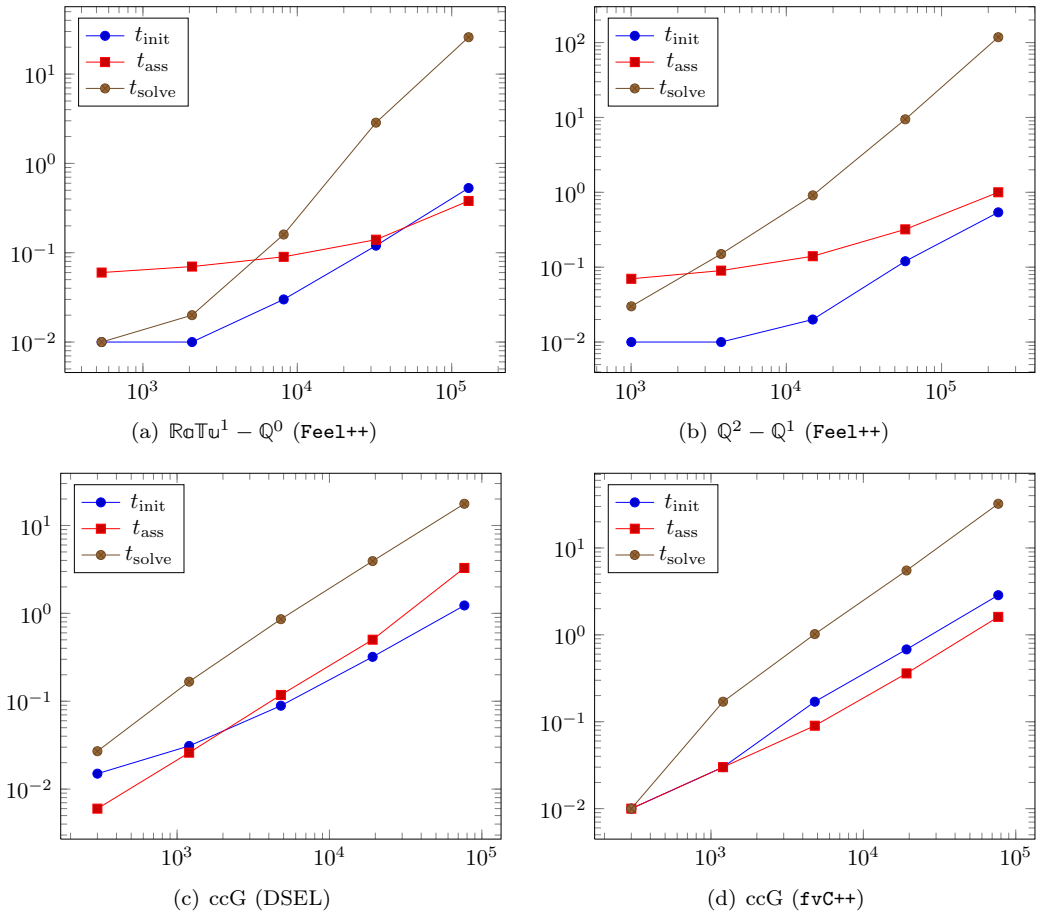
(g) $\|u - u_h\|_{L^2(\Omega)^d}$ vs. N_{nz} (h) $\|p - p_h\|_{L^2(\Omega)}$ vs. N_{nz} (i) $\mathcal{E}_{\mathrm{sto}}(u_h, p_h)$ vs. N_{nz}

FIG. 4.5. *Accuracy and memory consumption analysis for the example of Sect. 4.4*

The difficulty in this case is related to both the mesh, which mixes elongated triangular and quadrangular elements, and the anisotropy of the diffusion tensor. The results are presented in Figure 4.8. To facilitate the comparison with the results of [4, Figure 5], the discrete energy norm is defined according to [4, eq. (4.1)] for both the G-method and the SUSHI scheme, while for the ccG method we have used the norm of [15, eq. (3.7)]. While all of the methods have cell centered unknowns only, their stencils differ significantly. It is interesting to remark that, despite its larger stencil, the ccG method outperforms both the G-method and the SUSHI methods in terms of the discrete H^1-norm even when relating the error to the number of nonzero elements in the matrix. On the other hand, the G-method displays good convergence properties in the L^2-norm, but its performance is poor when it comes to the discrete H^1-norm. Finally, the SUSHI method may be a compromise when the memory occupation of the ccG method becomes inacceptable.

REFERENCES

[1] *ISO/IEC 14977*, 1996(E).
[2] I. Aavatsmark, T. Barkve, Ø. Bøe, and T. Mannseth, *Discretization on unstructured grids for inhomogeneous, anisotropic media, Part I: Derivation of the methods*, SIAM J. Sci. Comput., 19 (1998), pp. 1700–1716.
[3] I. Aavatsmark, G. T. Eigestad, B. T. Mallison, and J. M. Nordbotten, *A compact multipoint*



(a) $\mathbb{R}\mathbb{T}^1 - \mathbb{Q}^0$ (`Feel++`)

(b) $\mathbb{Q}^2 - \mathbb{Q}^1$ (`Feel++`)

(c) ccG (DSEL)

(d) ccG (`fvC++`)

FIG. 4.6. *Performance analysis for the example of Sect. 4.4*

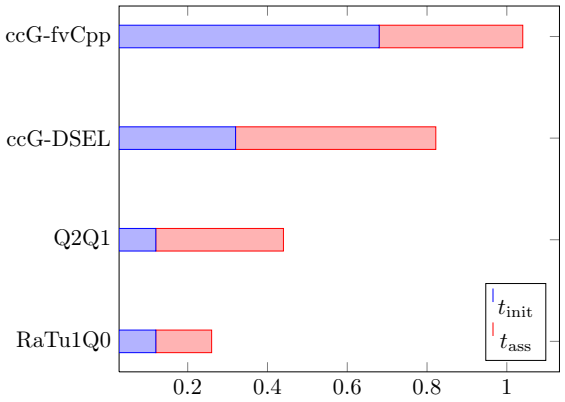


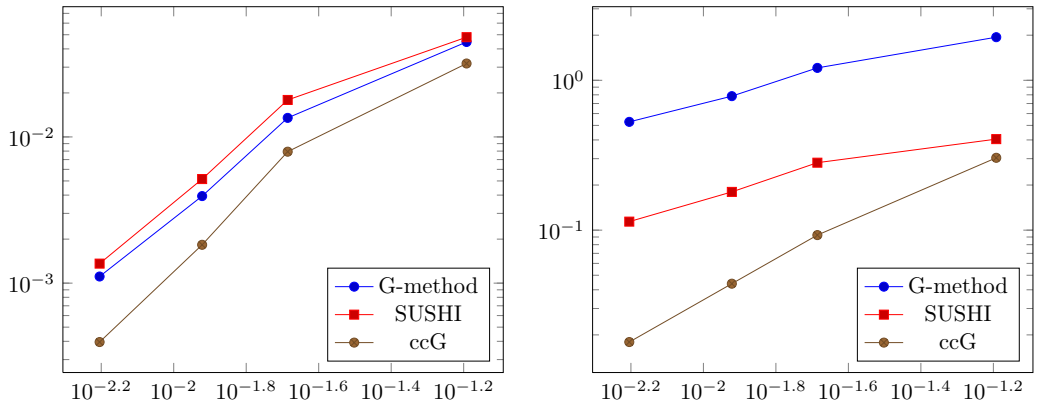
FIG. 4.7. *Comparison of different methods and implementations for the test case of Sect. 4.4*

flux approximation method with improved robustness, Numer. Methods Partial Differ. Eq., 24 (2008), pp. 1329–1360.

[4] L. AGÉLAS, D. A. DI PIETRO, AND J. DRONIOU, *The G method for heterogeneous anisotropic diffusion on general meshes*, M2AN Math. Model. Numer. Anal., 44 (2010), pp. 597–625.

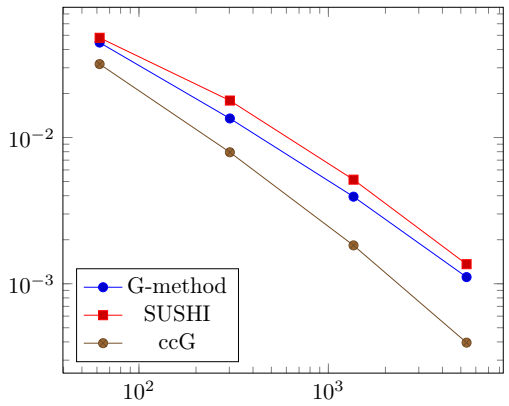
[5] L. AGÉLAS, D. A. DI PIETRO, R. EYMARD, AND R. MASSON, *An abstract analysis framework for nonconforming approximations of diffusion problems on general meshes*, IJFV, 7 (2010), pp. 1–29.

[6] D. N. ARNOLD, *An interior penalty finite element method with discontinuous elements*, SIAM J. Numer.

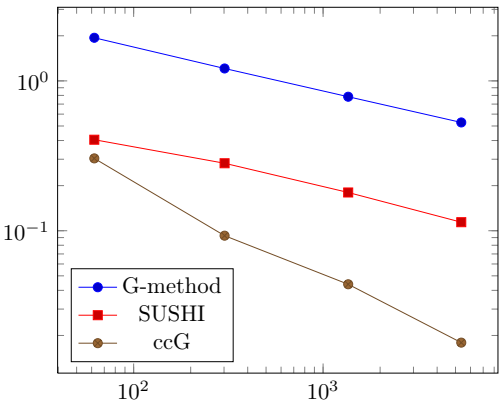


(a) L^2-error vs. h

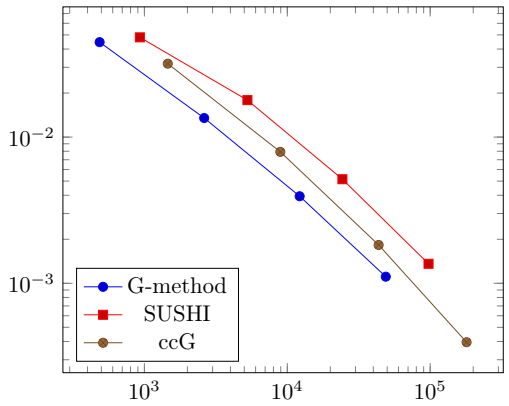
(b) H^1-error vs. h



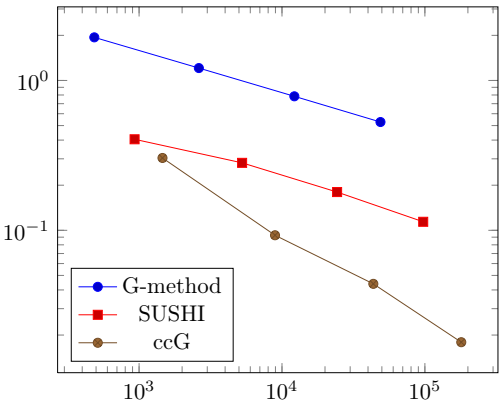
(c) L^2-error vs. N_{DOF}



(d) H^1-error vs. N_{DOF}



(e) L^2-error vs. N_{nz}



(f) H^1-error vs. N_{nz}

FIG. 4.8. *Accuracy and memory consumption analysis for the example of Sect. 4.5*

Anal., 19 (1982), pp. 742–760.

[7] F. BASSI, L. BOTTI, A. COLOMBO, D. A. DI PIETRO, AND P. TESINI, *On the flexibility of agglomeration based physical space discontinuous Galerkin discretizations*, J. Comput. Phys., 231 (2012), pp. 45–65.

[8] F. BREZZI, K. LIPNIKOV, AND M. SHASHKOV, *Convergence of mimetic finite difference methods for diffusion problems on polyhedral meshes*, SIAM J. Numer. Anal., 45 (2005), pp. 1872–1896.

[9] F. BREZZI, K. LIPNIKOV, AND V. SIMONCINI, *A family of mimetic finite difference methods on polygonal and polyhedral meshes*, M3AS, 15 (2005), pp. 1533–1553.

[10] E. BURMAN AND B. STAMM, *Minimal stabilization for discontinuous Galerkin finite element methods for*

hyperbolic problems, Communications in Computational Physics, (2009), pp. 498–524.
[11] P. G. Ciarlet, *Basic error estimates for elliptic problems*, in Handbook of Numerical Analysis, P. G. Ciarlet and J.-L-Lions, eds., vol. II: Finite Element Methods, North-Holland, Amsterdam, 1991, ch. 2.
[12] D. A. Di Pietro, *Analysis of a discontinuous Galerkin approximation of the Stokes problem based on an artificial compressibility flux*, Int. J. Numer. Methods Fluids, 55 (2007), pp. 793–813.
[13] ———, *Cell-centered Galerkin methods*, C. R. Math. Acad. Sci. Paris, 348 (2010), pp. 31–34.
[14] ———, *A compact cell-centered Galerkin method with subgrid stabilization*, C. R. Acad. Sci. Paris, Ser. I., 348 (2011), pp. 93–98.
[15] ———, *Cell centered Galerkin methods for diffusive problems*, M2AN Math. Model. Numer. Anal., 46 (2012), pp. 111–144.
[16] D. A. Di Pietro and A. Ern, *Analysis of a discontinuous Galerkin method for heterogeneous diffusion problems with low-regularity solutions*, Numer. Meth. PDEs, (2011). Published online. DOI: 10.1002/num.20675.
[17] ———, *Mathematical Aspects of Discontinuous Galkerin Methods*, no. 69 in Mathématiques & Applications, Springer Verlag, Berlin, 2011.
[18] D. A. Di Pietro, A. Ern, and J.-L. Guermond, *Discontinuous Galerkin methods for anisotropic semi-definite diffusion with advection*, SIAM J. Numer. Anal., 46 (2008), pp. 805–831.
[19] D. A. Di Pietro and J.-M. Gratien, *Lowest order methods for diffusive problems on general meshes: A unified approach to definition and implementation*, in Finite Volumes for Complex Applications VI, J. Fořt, J. Fürst, J. Halama, R. Herbin, and F. Hubert, eds., Springer–Verlag, 2011, pp. 3–19.
[20] J. Droniou and R. Eymard, *A mixed finite volume scheme for anisotropic diffusion problems on any grid*, Num. Math., 105 (2006), pp. 35–71.
[21] J. Droniou, R. Eymard, T. Gallouët, and R. Herbin, *A unified approach to mimetic finite difference, hybrid finite volume and mixed finite volume methods*, M3AS, 20 (2010), pp. 265–295.
[22] M.G. Edwards and C.F. Rogers, *Finite volume discretization with imposed flux continuity for the general tensor pressure equation*, Comput. Geosci., 2 (1998), pp. 259–290.
[23] A. Ern and J.-L. Guermond, *Theory and Practice of Finite Elements*, vol. 159 of Applied Mathematical Sciences, Springer-Verlag, New York, NY, 2004.
[24] R. Eymard, Th. Gallouët, and R. Herbin, *Discretization of heterogeneous and anisotropic diffusion problems on general nonconforming meshes SUSHI: a scheme using stabilization and hybrid interfaces*, IMA J. Numer. Anal., 30 (2010), pp. 1009–1043.
[25] R. Eymard, G. Henry, R. Herbin, F. Hubert, R. Klöfkorn, and G. Manzini, *3D benchmark on discretization schemes for anisotropic diffusion problems on general grids*, in Finite Volumes for Complex Applications VI Problems & Perspectives, J. Halama R. Herbin J. Fořt, J. Fürst and F. Hubert, eds., Springer-Verlag, 2011, pp. 95–130.
[26] C. Geuzaine and J.-F. Remacle, *Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities*, International Journal for Numerical Methods in Engineering, 79 (2009), pp. 1309–1331.
[27] G. Grospellier and B. Lelandais, *The Arcane development framework*, in Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing, POOSC '09, New York, NY, USA, 2009, ACM, pp. 4:1–4:11.
[28] P. Hansbo and M. G. Larson, *Discontinuous Galerkin and the Crouzeix–Raviart element: application to elasticity*, M2AN Math. Model. Numer. Anal., 1 (2003), pp. 63–72.
[29] F. Hecht and O. Pironneau, *FreeFEM++ Manual*, Laboratoire Jacques Louis Lions, 2005.
[30] R. Herbin and F. Hubert, *Benchmark on discretization schemes for anisotropic diffusion problems on general grids*, in Finite Volumes for Complex Applications V, John Wiley & Sons, 2008, pp. 659–692.
[31] A. Logg, J. Hoffman, R.C. Kirby, and J. Jansson, *Fenics*. `http://www.fenics.org/`, 2005.
[32] E. Niebler, `boost::proto` *documentation*, 2011. `http://www.boost.org/doc/libs/1_47_0/doc/html/proto.html`.
[33] C. Prud'homme, *A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations*, Scientific Programming, 2 (2006), pp. 81–110.
[34] C. Prud'homme, V. Chabannes, and G. Pena, *Feel++: A computational framework for Galerkin methods*, (2012). In preparation.
[35] C. Prud'homme, V. Chabannes, G. Pena, and S. Veys, *Feel++: Finite Element Embedded Language in C++*. Free Software available at `http://www.feelpp.org`. Contributions from A. Samake, V. Doyeux, M. Ismail.

TABLE 3.1
Mesh accessors for an object `Th` *of type* `Mesh`

Item set	Accessor
$\mathcal{T}_h$	`allCells(Th)`
$\mathcal{F}_h$	`allFaces(Th)`
$\mathcal{F}_h^i$	`interfaces(Th)`
$\mathcal{F}_h^b$	`boundaryFaces(Th)`

TABLE 3.2
Template parameters for the gradient reconstructions operator of Sect. 2

Name	Definition	Type name	**submesh**	`interpolator`	`dof`
$\mathfrak{G}_h^{\mathrm{g}}$	(2.9)	`GGradient`	`Pyramidal`	`Barycentric`	`CellCentered`
$\mathfrak{G}_h^{\mathrm{green}}$	(2.12)	`GreenGradient`	`Identity`	`LInterpolator`	`CellCentered`
$\mathfrak{G}_h^{\mathrm{hyb}}$	(2.17)	`SUSHIHGradient`	`Pyramidal`	`NoInterpolator`	`Hybrid`
$\mathfrak{G}_h^{\mathrm{cc}}$	(2.20)	`SUSHICCGradient`	`Pyramidal`	`LInterpolator`	`CellCentered`

TABLE 3.3
Template parameters for the discrete spaces of Sect. 2

Name	$\mathcal{S}_h$	**poly**	**gradient**
$\mathbb{P}_d^0(\mathcal{T}_h)$	`Identity`	**poly**`<0>`	**gradient**`<NullGradient>`
V_h^{g}	`Pyramidal`	**poly**`<1>`	**gradient**`<GGradient>`
V_h^{ccg}	`Identity`	**poly**`<1>`	**gradient**`<GreenGradient>`
V_h^{hyb}	`Pyramidal`	**poly**`<1>`	**gradient**`<SUSHIHGradient>`
V_h^{cc}	`Pyramidal`	**poly**`<1>`	**gradient**`<SUSHICCGradient>`