Implementation of Discontinuous Skeletal methods on arbitrary-dimensional, polytopal meshes using generic programming

Matteo Cicuttin, Daniele Di Pietro, Alexandre Ern

Implementation of Discontinuous Skeletal methods on arbitrary-dimensional, polytopal meshes using generic programming

M. Cicuttin, D. A. Di Pietro, A. Ern

September 5, 2017

Abstract

Discontinuous Skeletal methods approximate the solution of boundary-value problems by attaching discrete unknowns to mesh faces (hence the term skeletal) while allowing these discrete unknowns to be chosen independently on each mesh face (hence the term discontinuous). Cell-based unknowns, which can be eliminated locally by a Schur complement technique (also known as static condensation), are also used in the formulation. Salient examples of high-order Discontinuous Skeletal methods are Hybridizable Discontinuous Galerkin methods and the recently-devised Hybrid High-Order methods. Some major benefits of Discontinuous Skeletal methods are that their construction is dimension-independent and that they offer the possibility to use general meshes with polytopal cells and non-matching interfaces. In this work, we show how this mathematical flexibility can be efficiently replicated in a numerical software using generic programming. We describe a number of generic algorithms and data structures for high-order Discontinuous Skeletal methods within a "write once, run on any kind of mesh" framework. The computational efficiency of the implementation is assessed on the Poisson model problem discretized using various polytopal meshes and the Hybrid High-Order method.

1 Introduction

Discontinuous Skeletal (DiSk) methods are based on discrete unknowns that are discontinuous polynomials on the mesh skeleton. Additional cell-based unknowns are generally considered as well in the formulation of the method; such unknowns can be eliminated locally by a Schur complement technique (often referred to as static condensation in the finite element context). Eliminating the cell-based unknowns then leads to a global transmission problem coupling the face-based unknowns by means of a local stencil involving only adjacent elements in the sense of faces. DiSk methods present several attractive features: the mathematical construction is dimension-independent, arbitrary polynomial orders can be used, and general meshes, including polytopal cells and non-matching interfaces, are supported. Positioning unknowns at mesh faces is also a natural way to express locally in each mesh cell the fundamental balance properties satisfied by the boundary-value problem at hand.

Seminal instances of lowest-order discretization methods placing unknowns at mesh faces are the Mimetic Finite Difference (MFD) methods (see [9] and the book [7]) and the Hybrid Finite Volume (HFV) methods (see [24] and the unifying approach with MFD in [22]); see also the Cell

Boundary Element (CBE) method in [29]. Perhaps one of the most well-known high-order DiSk methods are the Hybridizable Discontinuous Galerkin (HDG) methods introduced in [14] (see also the review in [11] and the references therein); we also mention the Generalized CBE method in [28]. Another prominent example of DiSk methods are the Hybrid High-Order (HHO) methods, which were recently introduced and analyzed in [16] for linear elasticity and in [18] for scalar diffusion, see also the mixed formulation in [17] and the recent review in [19]. HHO methods are currently undergoing a vigorous development, see, among others, their extensions to advection-diffusion [15], Stokes [20], Cahn–Hilliard [10], and Biot's poroelasticity [8] equations. HHO methods have been recently bridged to HDG methods in [12], where a numerical flux trace for HHO methods has been identified in the context of a diffusion model problem. The three major differences between HDG and HHO methods are that HHO methods (i) are derived directly from the primal formulation, (ii) reconstruct locally the flux in a smaller space (exploiting its representation as the gradient of a potential) and (iii) deploy a (rather subtle) stabilization which achieves higher-order convergence rates on general meshes even when using simple polynomial spaces for the discrete unknowns (in contrast, similar properties for HDG methods generally require an enrichment of the underlying polynomial spaces, see, e.g., the recent work in [13]). Other examples of recent methods placing unknowns at mesh faces are the nonconforming Virtual Element method in [3] (which has been bridged to the HHO method in [12]) and the Weak Galerkin method in [32] (which has been bridged to HDG in [11]).

DiSk methods are devised at the mathematical level in a dimension-independent and cell-shape-independent fashion. The implementation, at least in principle, should reproduce this feature: a single piece of code should be able to work in any space dimension and to deal with any cell shape. It is not common, however, to see software packages taking this approach. In the vast majority of the cases, the codes are capable to run only on few very specific kinds of mesh, or only in 1D or 2D or 3D. On the one hand, this can happen simply because a fully general tool is not always needed. On the other hand, the programming languages commonly used by the scientific computing community (in particular Fortran and Matlab) are not easily amenable to an implementation which is generic and efficient at the same time. The usual (and natural) approach, in conventional languages, is to have different versions of the code, for example one specialized for 1D, one for 2D and one for 3D applications, making the overall maintenance of the codes rather cumbersome. The same considerations generally apply to the handling of mesh cells with various shapes, i.e., codes written in conventional languages generally support only a limited (and set in advance) number of cell shapes.

Generic programming [2] offers a valuable tool to address the above issues. Different libraries are already using such an approach to different extents; we mention, among others, [4, 6, 21, 27, 30, 31]. What generic programming provides is the possibility to write code where the data types are not immediately specified, but they are left as parameters. For example, a sort function written in generic style will not make any assumption neither on the data it will sort nor on the structure that contains data. Generic programming can be used also in a numerical code: by writing the code generically, it is possible to avoid making any assumption neither on the dimension (1D, 2D, 3D) of the problem, nor on the kind of mesh. In some sense, writing generic code resembles writing pseudocode: the compiler will take care of giving the correct meaning to each basic operation. As a result, with generic programming there will still be different versions of the code, but they will be generated by the compiler, and not by the programmer. As these considerations suggest, generic programming is a static technique: if correctly realized, the abstractions do not penalize the performance at runtime, because they will leave no trace in the generated code.

In the present work, we discuss a set of generic tools that can be used to implement DiSk methods, with a particular focus on HHO methods. The tools include a generic data structure to represent the discrete geometry (the mesh) and a set of operations on it. The data structure, which is composed of a three-layer abstraction, allows the user to handle simply "cells" and "faces" without having to care about what they really are (since the answer to this question depends on the ambient space dimension and the shape of the cell). To achieve this goal, the first layer hides the low-level details of a particular mesh format, and allows one to manipulate different "shapes" without caring about where they did come from and how they were originally represented. The second layer of abstraction hides the information about the shape of the cells and faces, but preserves the information about the dimension. Finally, the third layer hides the information about the dimension, presenting to the user just the concepts of cell and face. At this point, no matter what cells and faces actually are, it is possible to manipulate them in a unified and consistent way, and to obtain, at the same time, fine-tuned executable code.

This paper is organized as follows. In Section 2, we present the HHO method that we use as our main example among DiSk methods. We recall the functional formulation of the HHO method and, as a more practically-oriented presentation, we also describe its algebraic formulation (see also [1] for an algebraic presentation of the hybridized mixed HHO method). For simplicity, we focus on a simple model elliptic problem: the Poisson problem posed on a polytopal domain of $\mathbb{R}^d$ with homogeneous Dirichlet boundary conditions. In Section 3, we describe a generic implementation of DiSk methods on arbitrary-dimensional polytopal meshes. We first consider the various levels of abstraction necessary to handle mesh data structures in this context. Then we describe how various queries on the mesh can be implemented in this framework. In Section 4, we present a profiling of the implementation of the HHO method using the approach of Section 3 on a model elliptic problem using various two- and three-dimensional polytopal meshes. Finally, in Section 5, some conclusions are drawn.

2 An example: The HHO method

The goal of this section is to illustrate both the functional and the algebraic formulations of the HHO method when used to approximate the Poisson problem with homogeneous Dirichlet boundary conditions. Let $\Omega \subset \mathbb{R}^d$, $d \geq 1$, be an open, bounded, connected, strongly Lipschitz set in $\mathbb{R}^d$. For simplicity, we assume that Ω is a polygon/polyhedron. We want to approximate the weak solution $u \in H^1_0(\Omega)$ such that

$$(\nabla u, \nabla v)_\Omega = (f, v)_\Omega, \qquad \forall v \in H^1_0(\Omega), \tag{1}$$

where $(\cdot,\cdot)_\Omega$ denotes , according to the context, the standard inner product of $L^2(\Omega)$ or $L^2(\Omega;\mathbb{R}^d)$, and we assume for simplicity that $f \in L^2(\Omega)$.

2.1 Functional formulation

This section describes the functional formulation of the HHO method to approximate the model problem (1); for other boundary conditions, we refer the reader to [19]. This formulation has been introduced in [16, 18] hinging on two key operators, the reconstruction and the stabilization operator that are recalled below. We denote by $\mathcal{T}$ a mesh of Ω consisting of open disjoint polygonal/polyhedral cells with planar faces such that $\overline{\Omega} = \bigcup_{T\in\mathcal{T}} \overline{T}$. A generic mesh cell is denoted

$T \in \mathcal{T}$. We say that any subset $F \subset \mathbb{R}^d$ is a mesh face if it is a subset with nonempty relative interior of some affine hyperplane H_F and if either there are two distinct mesh cells $T_1, T_2 \in \mathcal{T}$ so that $F = \partial T_1 \cap \partial T_2 \cap H_F$ and F is called an interface, or there is one mesh cell $T \in \mathcal{T}$ so that $F = \partial T \cap \partial\Omega \cap H_F$ and F is called a boundary face. By construction, mesh faces are closed subsets of $\mathbb{R}^d$ and have mutually disjoint interiors. The mesh faces are collected in the set $\mathcal{F}$, interfaces in the set $\mathcal{F}^{\mathrm{i}}$, and boundary faces in the set $\mathcal{F}^{\mathrm{b}}$. For all $T \in \mathcal{T}$, let $\mathcal{F}_{\partial T}$ collect the mesh faces that are subsets of ∂T. Note that we have $\{\mathbf{x} \in \partial T\} = \bigcup_{F \in \mathcal{F}_{\partial T}} \{\mathbf{x} \in F\}$ and $\bigcup_{T \in \mathcal{T}} \{\mathbf{x} \in \partial T\} = \bigcup_{F \in \mathcal{F}} \{\mathbf{x} \in F\}$.

In what follows, $(\cdot,\cdot)_T$ and $(\cdot,\cdot)_F$ denote the $L^2(T)$- and $L^2(F)$-inner products for all $T \in \mathcal{T}$ and all $F \in \mathcal{F}$, respectively. The same notation is used for the inner products of $L^2(T)^d$ and $L^2(F)^d$. Since DiSk methods only require mesh cells and faces (mesh edges and vertices are not needed), we generically denote the mesh as the pair $\mathcal{M} \coloneqq (\mathcal{T}, \mathcal{F})$.

2.1.1 Local polynomial spaces

Let us fix a polynomial degree $k \geq 0$. For each mesh cell $T \in \mathcal{T}$, let $\mathbb{P}_d^k(T)$ be the space composed of d-variate polynomials of degree at most k restricted to T. For each face $F \in \mathcal{F}_{\partial T}$, the space $\mathbb{P}_{d-1}^k(F)$ is composed of the restrictions to F of the polynomials in $\mathbb{P}_d^k(T)$. Since the face F is planar by assumption, this space can be described as $\mathbb{P}_{d-1}^k(F) = \mathbb{P}_{d-1}^k \circ \mathbf{T}_F^{-1}$ where $\mathbf{T}_F : \mathbb{R}^{d-1} \to H_F$ is an affine bijective mapping and where H_F is the affine hyperplane in $\mathbb{R}^d$ supporting the face F. The space $\mathbb{P}_{d-1}^k(F)$ is independent of the choice of $\mathbf{T}_F$; indeed, considering another affine bijective mapping $\hat{\mathbf{T}}_F : \mathbb{R}^{d-1} \to H_F$, one can observe that $\mathbf{T}_F^{-1} \circ \hat{\mathbf{T}}_F$ is an affine bijective mapping from $\mathbb{R}^{d-1}$ onto itself, so that $\mathbb{P}_{d-1}^k \circ (\mathbf{T}_F^{-1} \circ \hat{\mathbf{T}}_F) = \mathbb{P}_{d-1}^k$ and hence $\mathbb{P}_{d-1}^k \circ \mathbf{T}_F^{-1} = \mathbb{P}_{d-1}^k \circ \hat{\mathbf{T}}_F^{-1}$.

We define the piecewise polynomial space

$$\mathbb{P}_{d-1}^k(\mathcal{F}_{\partial T}) \coloneqq \bigtimes_{F \in \mathcal{F}_{\partial T}} \mathbb{P}_{d-1}^k(F), \tag{2}$$

and we observe that an element $v_{\partial T} \in \mathbb{P}_{d-1}^k(\mathcal{F}_{\partial T})$ is a collection of polynomials indexed by the faces of T, i.e., we have $v_{\partial T} = (v_F)_{F \in \mathcal{F}_{\partial T}}$ where $v_F \in \mathbb{P}_{d-1}^k(F)$ for all $F \in \mathcal{F}_{\partial T}$. Note that there is no matching condition enforced on the polynomials v_F at vertices (in 2D) or edges (in 3D) separating neighboring faces in $\mathcal{F}_{\partial T}$. Then, for each mesh cell $T \in \mathcal{T}$, the local discrete space is

$$U_T^k \coloneqq \mathbb{P}_d^k(T) \times \mathbb{P}_{d-1}^k(\mathcal{F}_{\partial T}). \tag{3}$$

2.1.2 Reconstruction operator

Let $T \in \mathcal{T}$ be a mesh cell. We define $\mathbb{P}_{*d}^{k+1}(T) \coloneqq \{q \in \mathbb{P}_d^{k+1}(T) \mid (q,1)_T = 0\}$ (any supplementary subspace of the subspace spanned by the constant function in $\mathbb{P}_d^{k+1}(T)$ can be equivalently considered). The local reconstruction operator $R_T^{k+1} : U_T^k \to \mathbb{P}_{*d}^{k+1}(T)$ is defined such that, for all $(v_T, v_{\partial T}) \in U_T^k$ and all $w \in \mathbb{P}_{*d}^{k+1}(T)$,

$$(\nabla R_T^{k+1}(v_T, v_{\partial T}), \nabla w)_T \coloneqq (\nabla v_T, \nabla w)_T + \sum_{F \in \mathcal{F}_{\partial T}} (v_{\partial T} - v_T, \nabla w \cdot \boldsymbol{n}_T)_F, \tag{4}$$

where $\boldsymbol{n}_T$ is the unit outward normal to T. By the Riesz representation theorem in $\nabla\mathbb{P}_{*d}^{k+1}(T)$ for the $L^2(T;\mathbb{R}^d)$-inner product, this uniquely defines $\nabla R_T^{k+1}(v_T, v_{\partial T})$ and, recalling the zero-average condition, also $R_T^{k+1}(v_T, v_{\partial T})$.

The local reconstruction operator R_T^{k+1} is used to build the following bilinear form on $U_T^k \times U_T^k$:

$$a_T^{(1)}((v_T, v_{\partial T}), (w_T, w_{\partial T})) = (\nabla R_T^{k+1}(v_T, v_{\partial T}), \nabla R_T^{k+1}(w_T, w_{\partial T}))_T, \tag{5}$$

which mimics locally the bilinear form in the left-hand side of (1).

One can show that the local reconstruction operator R_T^{k+1} enjoys a polynomial consistency property of order $(k+1)$. To formalize this property, let Π_T^k and Π_F^k denote the L^2-orthogonal projectors onto $\mathbb{P}_d^k(T)$ and $\mathbb{P}_{d-1}^k(F)$, respectively. Let us then define the reduction map $I_T^k : H^1(T) \to U_T^k$ so that, for a function $v \in H^1(T)$, the cell component of $I_T^k(v)$ is the cell L^2-orthogonal projection $\Pi_T^k(v)$ and the face components of $I_T^k(v)$ are the face L^2-orthogonal projections $\Pi_F^k(v)$, for all $F \in \mathcal{F}_T$. Then, the following consistency property holds:

$$\nabla R_T^{k+1}(I_T^k(q)) = \nabla q, \qquad \forall q \in \mathbb{P}_d^{k+1}(T). \tag{6}$$

2.1.3 Stabilization operator

For $(v_T, v_{\partial T}) \in U_T^k$, the reconstructed gradient $\nabla R_T^{k+1}(v_T, v_{\partial T})$ is not stable in the sense that $\nabla R_T^{k+1}(v_T, v_{\partial T}) = 0$ does not necessarily mean that v_T and $v_{\partial T}$ are constant functions taking the same value. To restore stability, we introduce a stabilization operator that relates the cell and face polynomials at the boundary of T. We define the stabilization operator $S_T^k : U_T^k \to \mathbb{P}_{d-1}^k(\mathcal{F}_{\partial T})$ such that, for all $(v_T, v_{\partial T}) \in U_T^k$, letting $r_T^{k+1} \in \mathbb{P}_d^{k+1}(T)$ be such that $\nabla r_T^{k+1} = \nabla R_T^{k+1}(v_T, v_{\partial T})$ and $(r_T^{k+1} - v_T, 1)_T = 0$, we have

$$S_T^k(v_T, v_{\partial T}) \coloneqq \Pi_{\partial T}^k \left(v_{\partial T} - (v_T + r_T^{k+1} - \Pi_T^k r_T^{k+1})|_{\partial T} \right), \tag{7}$$

where $\Pi_{\partial T}^k$ is the L^2-orthogonal projector onto $\mathbb{P}_{d-1}^k(\mathcal{F}_{\partial T})$ acting on each face $F \in \mathcal{F}_{\partial T}$ as the projector Π_F^k defined above. Using this stabilization operator, we build the following bilinear form on $U_T^k \times U_T^k$:

$$a_T^{(2)}((v_T, v_{\partial T}), (w_T, w_{\partial T})) = \sum_{F \in \mathcal{F}_{\partial T}} h_F^{-1} (S_T^k(v_T, v_{\partial T}), S_T^k(w_T, w_{\partial T}))_F, \tag{8}$$

where h_F denotes the diameter of the face F.

The design of the stabilization operator S_T^k by means of (7) ensures the following polynomial consistency property of order $(k+1)$:

$$S_T^k(I_T^k(q)) = 0, \qquad \forall q \in \mathbb{P}_d^{k+1}(T). \tag{9}$$

At the same time, the stabilization operator S_T^k achieves the following crucial stability property: Defining on U_T^k the semi-norm

$$|(v_T, v_{\partial T})|_{1,T}^2 \coloneqq \|\nabla v_T\|_T^2 + \sum_{F \in \mathcal{F}_{\partial T}} h_F^{-1} \|v_{\partial T} - v_T\|_F^2, \tag{10}$$

(so that $|(v_T, v_{\partial T})|_{1,T} = 0$ if and only if v_T and $v_{\partial T}$ are constant functions taking the same value), one can show that there is a uniform constant $c_1 > 0$ so that

$$c_1 |(v_T, v_{\partial T})|_{1,T}^2 \le \|\nabla R_T^{k+1}(v_T, v_{\partial T})\|_T^2 + \sum_{F \in \mathcal{F}_{\partial T}} h_F^{-1} \|S_T^k(v_T, v_{\partial T})\|_F^2, \tag{11}$$

for all $T \in \mathcal{T}$ and all $(v_T, v_{\partial T}) \in U_T^k$. Finally, we notice that the simpler definition $S_T^k(v_T, v_{\partial T}) \coloneqq \Pi_{\partial T}^k(v_{\partial T} - v_T|_{\partial T})$ for the stabilization operator leads to $S_T^k(I_T^k(v)) = \Pi_{\partial T}^k(v) - \Pi_T^k(v)|_{\partial T}$ for any function $v \in H^1(T)$ and thus only ensures the polynomial consistency property $S_T^k(I_T^k(q)) = 0$ for all $q \in \mathbb{P}_d^k(T)$.

2.1.4 Discrete problem

Recall the notation $\mathcal{M} = (\mathcal{T}, \mathcal{F})$ for the mesh. The local discrete spaces U_T^k, for all $T \in \mathcal{T}$, are collected into a global discrete space

$$U_{\mathcal{M}}^k \coloneqq U_{\mathcal{T}}^k \times U_{\mathcal{F}}^k, \tag{12}$$

where

$$U_{\mathcal{T}}^k \coloneqq \mathbb{P}_d^k(\mathcal{T}) \coloneqq \{v_{\mathcal{T}} = (v_T)_{T\in\mathcal{T}} \mid v_T \in \mathbb{P}_d^k(T),\ \forall T \in \mathcal{T}\}, \tag{13a}$$

$$U_{\mathcal{F}}^k \coloneqq \mathbb{P}_{d-1}^k(\mathcal{F}) \coloneqq \{v_{\mathcal{F}} = (v_F)_{F\in\mathcal{F}} \mid v_F \in \mathbb{P}_{d-1}^k(F),\ \forall F \in \mathcal{F}\}. \tag{13b}$$

Given a pair $v_{\mathcal{M}} \coloneqq (v_{\mathcal{T}}, v_{\mathcal{F}})$ in the global discrete space $U_{\mathcal{M}}^k$, for all $T \in \mathcal{T}$, we denote by $(v_T, v_{\partial T})$ its restriction to the local discrete space U_T^k, where $v_{\partial T} = (v_F)_{F\in\mathcal{F}_{\partial T}}$. We enforce strongly the homogeneous Dirichlet boundary condition by considering the subspace

$$U_{\mathcal{M},0}^k \coloneqq U_{\mathcal{T}}^k \times U_{\mathcal{F},0}^k, \tag{14}$$

where

$$U_{\mathcal{F},0}^k \coloneqq \{v_{\mathcal{F}} \in U_{\mathcal{F}}^k \mid v_F \equiv 0,\ \forall F \in \mathcal{F}^{\mathrm{b}}\}. \tag{15}$$

For all $T \in \mathcal{T}$, we combine the bilinear forms built using the reconstruction and the stabilization operators into a single bilinear form a_T on $U_T^k \times U_T^k$ such that

$$a_T \coloneqq a_T^{(1)} + a_T^{(2)}. \tag{16}$$

The discrete problem consists in seeking $u_{\mathcal{M}} \coloneqq (u_{\mathcal{T}}, u_{\mathcal{F}}) \in U_{\mathcal{M},0}^k$ such that

$$a_{\mathcal{M}}(u_{\mathcal{M}}, w_{\mathcal{M}}) = \ell_{\mathcal{M}}(w_{\mathcal{M}}), \qquad \forall w_{\mathcal{M}} \coloneqq (w_{\mathcal{T}}, w_{\mathcal{F}}) \in U_{\mathcal{M},0}^k, \tag{17}$$

where

$$a_{\mathcal{M}}(u_{\mathcal{M}}, w_{\mathcal{M}}) \coloneqq \sum_{T\in\mathcal{T}} a_T((u_T, u_{\partial T}), (w_T, w_{\partial T})), \tag{18a}$$

$$\ell_{\mathcal{M}}(w_{\mathcal{M}}) \coloneqq \sum_{T\in\mathcal{T}} (f, w_T)_T. \tag{18b}$$

The construction of the discrete problem (17) corresponds to a standard cellwise assembly as in finite element methods. The convergence analysis performed in [16, 18] leads to energy-error estimates of order h^{k+1} and to L^2-error estimates of order h^{k+2} if full elliptic regularity holds for the model problem.

Referring, e.g., to [19] for more details, we observe that the discrete problem (17) can be efficiently solved in a two-step procedure. In the first step, one expresses the cell unknowns $u_{\mathcal{T}}$ in terms of the face unknowns $u_{\mathcal{F}}$ and the source term f by solving the following coercive problem: Find $u_{\mathcal{T}} \in U_{\mathcal{T}}^k$ such that

$$a_{\mathcal{M}}((u_{\mathcal{T}}, 0), (w_{\mathcal{T}}, 0)) = \ell_{\mathcal{M}}(w_{\mathcal{T}}, 0) - a_{\mathcal{M}}((0, u_{\mathcal{F}}), (w_{\mathcal{T}}, 0)), \tag{19}$$

for all $w_{\mathcal{T}} \in U_{\mathcal{T}}^k$. This problem is easy to solve since the unknowns attached to distinct mesh cells remain uncoupled, which is reflected by the fact that the matrix in the left-hand side is block

diagonal; see the discussion in Section 2.2.4. In the second step, one solves the following coercive problem: Find $u_{\mathcal{F}} \in U_{\mathcal{F}}^k$ such that

$$a_{\mathcal{M}}((0,u_{\mathcal{F}}),(0,w_{\mathcal{F}})) = a_{\mathcal{M}}((u_{\mathcal{T}},0),(0,w_{\mathcal{F}})), \tag{20}$$

for all $w_{\mathcal{F}} \in U_{\mathcal{F}}^k$, where $u_{\mathcal{T}} \in U_{\mathcal{T}}^k$ results from (19). One can show that the discrete problem (20) is a global transmission problem that expresses the fact that, across each interface $F \in \mathcal{F}^{\mathrm{i}}$, the jump of the numerical normal flux trace is zero.

2.2 Algebraic formulation

In this section we introduce the local matrices $\mathbf{A}_T^{(1)}$ and $\mathbf{A}_T^{(2)}$ that are the algebraic counterparts of the bilinear forms $a_T^{(1)}$ and $a_T^{(2)}$ introduced above. The implementation of these matrices is further described in Algorithm 1 from this Subsection and in the Listings 7 and 8 from Subsection 3.2. Moreover, the entries of these matrices are computed with quadratures that are further described in Algorithm 2 from Subsection 3.2.2.

In what follows, we adopt the convention that the indexing of vectors and matrices starts from 0. For integers $l \geq 0$ and $n \geq 0$, we denote by $N_n^l := \binom{l+n}{l}$ the dimension of the space composed of n-variate polynomials of degree at most l. In what follows, we shall need the numbers N_d^k, N_{d-1}^k, and N_d^{k+1}, where k is the polynomial degree used in the HHO method and d is the space dimension.

2.2.1 Local basis functions

Let $T \in \mathcal{T}$ be a mesh cell. We fix a basis $\{\phi_{T,i}\}_{0\leq i<N_d^{k+1}}$ of $\mathbb{P}_d^{k+1}(T)$. To simplify the presentation, we assume that the basis functions are such that (i) $\phi_{T,0}$ is the constant function (i.e., this function spans the kernel of the gradient operator), (ii) $\{\phi_{T,i}\}_{0\leq i<N_d^k}$ is a basis of $\mathbb{P}_d^k(T)$. Letting $N_{\partial T}$ denote the number of faces composing the boundary of T, we enumerate these faces from 0 to $N_{\partial T}-1$. For all $0 \leq m < N_{\partial T}$, we fix a basis $\{\psi_{F_m,n}\}_{0\leq n<N_{d-1}^k}$ of $\mathbb{P}_{d-1}^k(F_m)$; it is natural to set $\psi_{F_m,n} = \hat{\psi}_n \circ T_{F_m}^{-1}$, where $\{\hat{\psi}_n\}_{0\leq n<N_{d-1}^k}$ is a basis of $\mathbb{P}_{d-1}^k$ and $T_{F_m} : \mathbb{R}^{d-1} \to H_{F_m}$ is the above-introduced affine bijective mapping from $\mathbb{R}^{d-1}$ to the affine hyperplane H_{F_m} supporting F_m in $\mathbb{R}^d$.

Then, a pair $(u_T, u_{\partial T}) \in U_T^k$ can be viewed as a vector $\mathsf{U}_T \in \mathbb{R}^{N_T^k}$, with

$$N_T^k := N_d^k + N_{\partial T} \times N_{d-1}^k, \tag{21}$$

in such a way that

$$u_T = \sum_{0\leq i<N_d^k} \mathsf{U}_{T,i}\phi_{T,i}, \tag{22}$$

and $u_{\partial T} = (u_{F_m})_{0\leq m<N_{\partial T}}$ where, for all $0 \leq m < N_{\partial T}$,

$$u_{F_m} = \sum_{0\leq n<N_{d-1}^k} \mathsf{U}_{T,N_d^k+mN_{d-1}^k+n}\psi_{F_m,n}, \tag{23}$$

i.e., we order first the components of the polynomial attached to the cell and then, enumerating the faces composing the boundary of T, we order the components of the polynomial attached to each face.

We define the cell mass matrix $\mathbf{M}_T \in \mathbb{R}^{N_d^k\times N_d^k}$ such that

$$\mathbf{M}_{T,ij} := (\phi_{T,i},\phi_{T,j})_T, \qquad 0 \leq i,j < N_d^k, \tag{24}$$

and for all $0 \le m < N_{\partial T}$, the face mass matrix $\mathbf{M}_{F_m} \in \mathbb{R}^{N_{d-1}^k \times N_{d-1}^k}$ such that

$$\mathbf{M}_{F_m,nn'} \coloneqq (\psi_{F_m,n}, \psi_{F_m,n'})_{F_m}, \qquad 0 \le n, n' < N_{d-1}^k. \tag{25}$$

All of the above mass matrices are symmetric positive definite. Finally, let us set $N_{*d}^{k+1} \coloneqq N_d^{k+1} - 1$. We define the stiffness matrix $\mathbf{K}_T \in \mathbb{R}^{N_{*d}^{k+1} \times N_{*d}^{k+1}}$ such that

$$\mathbf{K}_{T,ij} \coloneqq (\nabla\phi_{T,i+1}, \nabla\phi_{T,j+1})_T, \qquad 0 \le i, j < N_{*d}^{k+1}. \tag{26}$$

Note that also $\mathbf{K}_T$ is a symmetric positive definite matrix.

The implementation of the HHO method requires the inversion of the local stiffness matrix $\mathbf{K}_T$ to evaluate the reconstruction operator and of the local matrices $\mathbf{M}_T$ and $\mathbf{M}_{F_m}$, $0 \le m < N_{\partial T}$, to evaluate the stabilization operator. These operations can be efficiently and robustly accomplished using the Cholesky algorithm. Notice that the cost of computing the Cholesky factorization of the mass matrices (an operation required to compute the stabilization term) is basically negligible with respect to the cost of factorizing the stiffness matrix for the computation of the reconstruction term. As a matter of fact, the factorization cost scales as the cube of the matrix size, and the mass matrices are related to polynomials of order k whereas the stiffness matrix is related to polynomials of order $(k+1)$.

2.2.2 Reconstruction operator

We define the rectangular matrix $\mathbf{V}_T \in \mathbb{R}^{N_{*d}^{k+1} \times N_T^k}$ (whose block structure is depicted in Figure 1) such that, for all $0 \le i < N_{*d}^{k+1}$ and all $0 \le j < N_d^k$,

$$\mathbf{V}_{T,ij} \coloneqq (\nabla\phi_{T,j}, \nabla\phi_{T,i+1})_T - \sum_{0 \le m < N_{\partial T}} (\phi_{T,j}, \nabla\phi_{T,i+1}\cdot\boldsymbol{n}_T)_{F_m}, \tag{27}$$

and for all $0 \le i < N_{*d}^{k+1}$ and all $N_d^k \le j < N_T^k$,

$$\mathbf{V}_{T,ij} \coloneqq (\psi_{F_m,n}, \nabla\phi_{T,i+1}\cdot\boldsymbol{n}_T)_{F_m}, \tag{28}$$

where $0 \le m < N_{\partial T}$ and $0 \le n < N_{d-1}^k$ are uniquely defined by the relation $j = N_d^k + mN_{d-1}^k + n$. Then, defining the symmetric positive semidefinite matrix (whose block structure is depicted in Figure 2)

$$\mathbf{A}_T^{(1)} \coloneqq \mathbf{V}_T^\dagger \mathbf{K}_T^{-1} \mathbf{V}_T, \qquad \mathbf{A}_T^{(1)} \in \mathbb{R}^{N_T^k \times N_T^k}, \tag{29}$$

where the superscript $\dagger$ denotes the transpose of a matrix, we infer that

$$a_T^{(1)}((v_T, v_{\partial T}), (w_T, w_{\partial T})) = \mathsf{W}_T^\dagger \mathbf{A}_T^{(1)} \mathsf{V}_T, \tag{30}$$

for all $(v_T, v_{\partial T}), (w_T, w_{\partial T}) \in U_T^k$ with components collected in the vectors V_T and W_T, respectively. In the implementation, the inverse of $\mathbf{K}_T$ is not computed, but only its Cholesky decomposition. The pseudocode describing the computation of the matrix $\mathbf{A}_T^{(1)}$ is detailed in Algorithm 1.

2.2.3 Stabilization operator

We define the rectangular matrix $\mathbf{N}_T \in \mathbb{R}^{N_d^k \times N_{*d}^{k+1}}$ such that

$$\mathbf{N}_{T,ij} \coloneqq (\phi_{T,i}, \phi_{T,j+1})_T, \qquad 0 \le i < N_d^k,\ 0 \le j < N_{*d}^{k+1}, \tag{31}$$

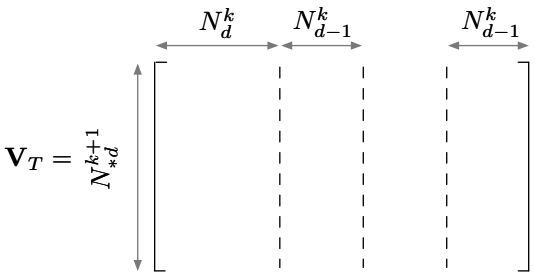

Figure 1: Graphical representation of the block structure of the matrix $\mathbf{V}_T$. It consists in a first column related to the interior of an element and of a variable number of additional columns related to the faces of the same element.

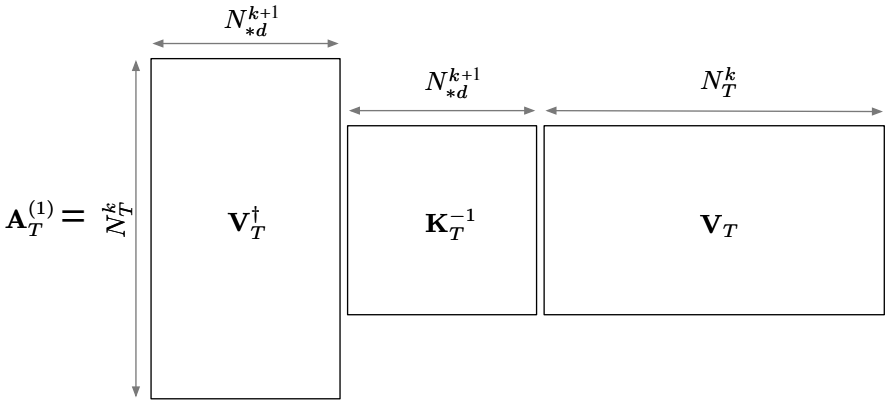

Figure 2: Graphical representation of the computation of the matrix $\mathbf{A}_T^{(1)}$. The product $\mathbf{K}_T^{-1}\mathbf{V}_T$ is carried out with a Cholesky factorization and a subsequent multi-rhs solve.

Algorithm 1 Procedure used to compute the matrix of the reconstruction operator. We denote with *integrate*() a pseudocode function that returns a list of pairs (Q_p, Q_w) of quadrature points Q_p and weights Q_w.

function BUILDRECONSTRUCTIONOPERATOR(T)
  initialize $\mathbf{V}_T \in \mathbb{R}^{N_{*d}^{k+1} \times N_T^k}$ with zeros
  $Q \leftarrow integrate(T)$
  for $(Q_p, Q_w) \in Q$ **do**
    for $0 \leq i < N_{*d}^{k+1}$ **do**
      for $0 \leq j < N_d^k$ **do**
        $\mathbf{V}_{T,ij} \leftarrow \mathbf{V}_{T,ij} + Q_w \nabla \phi_{T,j}(Q_p) \cdot \nabla \phi_{T,i+1}(Q_p)$
  for $F_m \in faces(T)$ **do**
    $Q = integrate(F_m)$
    for $(Q_p, Q_w) \in Q$ **do**
      for $0 \leq i < N_{*d}^{k+1}$ **do**
        for $0 \leq j < N_d^k$ **do**
          $\mathbf{V}_{T,ij} \leftarrow \mathbf{V}_{T,ij} + Q_w \nabla \phi_{T,j}(Q_p) \cdot \nabla \phi_{T,i+1}(Q_p)$
      for $0 \leq i < N_{*d}^{k+1}$ **do**
        for $0 \leq n < N_{d-1}^k$ **do**
          $j \leftarrow N_d^k + m N_{d-1}^k + n$
          $\mathbf{V}_{T,ij} \leftarrow \mathbf{V}_{T,ij} + Q_w \psi_{F_m,n}(Q_p) \nabla \phi_{T,i+1}(Q_p) \cdot \boldsymbol{n}_T$
  $\mathbf{R}_T \leftarrow \mathbf{K}_T^{-1} \mathbf{V}_T$
  $\mathbf{A}_T^{(1)} \leftarrow \mathbf{V}_T^\dagger \mathbf{K}_T^{-1} \mathbf{V}_T$

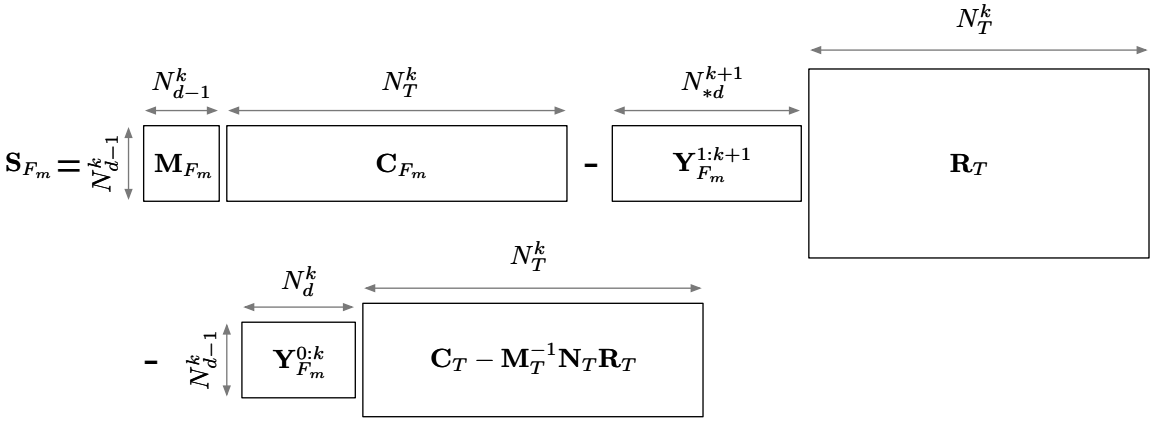

Figure 3: Graphical representation of the computation of the matrix $\mathbf{S}_{F_m}$.

and the rectangular matrices $\mathbf{Y}_{F_m} \in \mathbb{R}^{N_{d-1}^k \times N_d^{k+1}}$, for all $0 \le m < N_{\partial T}$, such that

$$\mathbf{Y}_{F_m,nj} \coloneqq (\psi_{F_m,n}, \phi_{T,j})_{F_m}, \qquad 0 \le n < N_{d-1}^k,\ 0 \le j < N_d^{k+1}. \tag{32}$$

It is convenient to extract the submatrices $\mathbf{Y}_{F_m,nj}^{0:k} \in \mathbb{R}^{N_{d-1}^k \times N_d^k}$ and $\mathbf{Y}_{F_m,nj}^{1:k+1} \in \mathbb{R}^{N_{d-1}^k \times N_{*d}^{k+1}}$ such that

$$\mathbf{Y}_{F_m,nj}^{0:k} \coloneqq \mathbf{Y}_{F_m,nj}, \qquad 0 \le n < N_{d-1}^k,\ 0 \le j < N_d^k, \tag{33a}$$

$$\mathbf{Y}_{F_m,nj}^{1:k+1} \coloneqq \mathbf{Y}_{F_m,n(j+1)}, \qquad 0 \le n < N_{d-1}^k,\ 0 \le j < N_{*d}^{k+1}. \tag{33b}$$

We also need the component matrices $\mathbf{C}_T \in \mathbb{R}^{N_d^k \times N_T^k}$ such that

$$\mathbf{C}_{T,ij} \coloneqq \delta_{ij}, \qquad 0 \le i < N_d^k,\ 0 \le j < N_T^k, \tag{34}$$

and $\mathbf{C}_{F_m} \in \mathbb{R}^{N_{d-1}^k \times N_T^k}$, $0 \le m < N_{\partial T}$, such that

$$\mathbf{C}_{F_m,nj} \coloneqq \delta_{n,j-N_d^k-mN_{d-1}^k}, \qquad 0 \le n < N_{d-1}^k,\ 0 \le j < N_T^k, \tag{35}$$

where the δ's are Kronecker symbols. These matrices simply recover the components attached to the cell or those attached to a specific face from the full set of components of a vector in $\mathbb{R}^{N_T^k}$. Finally, setting $\mathbf{R}_T \coloneqq \mathbf{K}_T^{-1}\mathbf{V}_T \in \mathbb{R}^{N_{*d}^{k+1} \times N_T^k}$ for the matrix associated with the reconstruction operator, we define, for all $0 \le m < N_{\partial T}$,

$$\mathbf{S}_{F_m} \coloneqq \mathbf{M}_{F_m}\mathbf{C}_{F_m} - \mathbf{Y}_{F_m}^{1:k+1}\mathbf{R}_T - \mathbf{Y}_{F_m}^{0:k}(\mathbf{C}_T - \mathbf{M}_T^{-1}\mathbf{N}_T\mathbf{R}_T). \tag{36}$$

The sequence of computations needed to compute $\mathbf{S}_{F_m}$ is depicted in Figure 3. Introducing the symmetric positive semidefinite matrix

$$\mathbf{A}_T^{(2)} \coloneqq \sum_{0 \le m < N_{\partial T}} h_{F_m}^{-1}\mathbf{S}_{F_m}^\dagger \mathbf{M}_{F_m}^{-1}\mathbf{S}_{F_m}, \qquad \mathbf{A}_T^{(2)} \in \mathbb{R}^{N_T^k \times N_T^k}, \tag{37}$$

we infer that

$$a_T^{(2)}((v_T, v_{\partial T}), (w_T, w_{\partial T})) = \mathsf{W}_T^\dagger \mathbf{A}_T^{(2)} \mathsf{V}_T, \tag{38}$$

for all $(v_T, v_{\partial T}), (w_T, w_{\partial T}) \in U_T^k$ with components collected in the vectors V_T and W_T, respectively.

2.2.4 Discrete problem

For any pair $v_{\mathcal{M}} = (v_{\mathcal{T}}, v_{\mathcal{F}})$ in the global discrete space $U^k_{\mathcal{M},0}$, its components using the polynomial bases attached to the mesh cells and faces are collected in a global component vector $\mathsf{V}_{\mathcal{M}} \in \mathbb{R}^{N^k_{\mathcal{M},0}}$ with

$$N^k_{\mathcal{M},0} \coloneqq \dim(U^k_{\mathcal{M},0}) = N_{\mathcal{T}} \times N^k_d + N_{\mathcal{F}^{\mathrm{i}}} \times N^k_{d-1}, \tag{39}$$

where $N_{\mathcal{T}}$ denotes the number of mesh cells and $N_{\mathcal{F}^{\mathrm{i}}}$ the number of mesh interfaces. Then, the algebraic realization of the discrete problem (17) is the linear system

$$\mathbf{A}_{\mathcal{M}}\mathsf{U}_{\mathcal{M}} = \mathsf{B}_{\mathcal{M}}, \tag{40}$$

where the unknown is the vector $\mathsf{U}_{\mathcal{M}} \in \mathbb{R}^{N^k_{\mathcal{M},0}}$ collecting the components of the discrete solution $u_{\mathcal{M}}$, the system matrix $\mathbf{A}_{\mathcal{M}}$ is assembled in the usual finite element fashion so that

$$\mathbf{A}_{\mathcal{M}} = \sum_{T\in\mathcal{T}} \mathbf{P}_T^\dagger(\mathbf{A}_T^{(1)} + \mathbf{A}_T^{(2)})\mathbf{P}_T, \tag{41}$$

with the local matrices $\mathbf{A}_T^{(1)}$ and $\mathbf{A}_T^{(2)}$ defined in (29) and (37), respectively, while the restriction matrix $\mathbf{P}_T \in \mathbb{R}^{N^k_T \times N^k_{\mathcal{M},0}}$ collects the components of a vector attached to a given mesh cell $T \in \mathcal{T}$ (and inserting zeros for components attached to possible boundary faces), and the right-hand side vector $\mathsf{B}_{\mathcal{M}}$ is constructed in a similar manner from the linear form $\ell_{\mathcal{M}}$ in (17).

As above, an efficient way of solving the linear system (40) consists of using a two-step procedure based on a Schur complement. Let us order for simplicity the components attached to cell unknowns first and then those attached to face unknowns. This induces the decomposition $\mathsf{U}_{\mathcal{M}} = (\mathsf{U}_{\mathcal{T}}, \mathsf{U}_{\mathcal{F}^{\mathrm{i}}})^\dagger$ with $\mathsf{U}_{\mathcal{T}} \in \mathbb{R}^{N^k_{\mathcal{T}}}$ with $N^k_{\mathcal{T}} = N_{\mathcal{T}} \times N^k_d$, and $\mathsf{U}_{\mathcal{F}^{\mathrm{i}}} \in \mathbb{R}^{N^k_{\mathcal{F}^{\mathrm{i}}}}$ with $N^k_{\mathcal{F}^{\mathrm{i}}} = N_{\mathcal{F}^{\mathrm{i}}} \times N^k_{d-1}$. Similarly, the decomposition of the right-hand side vector is $\mathsf{B}_{\mathcal{M}} = (\mathsf{B}_{\mathcal{T}}, 0)^\dagger$ and that of the system matrix leads (with obvious notation) to four blocks $\mathbf{A}_{\mathcal{T}\mathcal{T}}$, $\mathbf{A}_{\mathcal{T}\mathcal{F}^{\mathrm{i}}}$, $\mathbf{A}_{\mathcal{F}^{\mathrm{i}}\mathcal{T}}$, and $\mathbf{A}_{\mathcal{F}^{\mathrm{i}}\mathcal{F}^{\mathrm{i}}}$. Introducing a similar block decompositions for the right-hand side vector $\mathsf{B}_{\mathcal{M}}$, the system (40) rewrites

$$\left[\begin{array}{c:c} \mathbf{A}_{\mathcal{T}\mathcal{T}} & \mathbf{A}_{\mathcal{T}\mathcal{F}^{\mathrm{i}}} \\ \hdashline \mathbf{A}_{\mathcal{F}^{\mathrm{i}}\mathcal{T}} & \mathbf{A}_{\mathcal{F}^{\mathrm{i}}\mathcal{F}^{\mathrm{i}}} \end{array}\right]\left[\begin{array}{c} \mathsf{U}_{\mathcal{T}} \\ \hdashline \mathsf{U}_{\mathcal{F}^{\mathrm{i}}} \end{array}\right] = \left[\begin{array}{c} \mathsf{B}_{\mathcal{T}} \\ \hdashline 0 \end{array}\right]. \tag{42}$$

Then, the algebraic realization of the cell-based problem (19) reads

$$\mathbf{A}_{\mathcal{T}\mathcal{T}}\mathsf{U}_{\mathcal{T}} = \mathsf{B}_{\mathcal{T}} - \mathbf{A}_{\mathcal{T}\mathcal{F}^{\mathrm{i}}}\mathsf{U}_{\mathcal{F}^{\mathrm{i}}}, \tag{43}$$

where the submatrix $\mathbf{A}_{\mathcal{T}\mathcal{T}}$ is block-diagonal (each block having size N^k_d), and using (43), the algebraic realization of the global transmission problem (19) becomes

$$\mathbf{A}_{\mathrm{S}}\mathsf{U}_{\mathcal{F}^{\mathrm{i}}} = -\mathbf{A}_{\mathcal{F}^{\mathrm{i}}\mathcal{T}}\mathbf{A}^{-1}_{\mathcal{T}\mathcal{T}}\mathsf{B}_{\mathcal{T}}, \tag{44}$$

with the Schur complement matrix

$$\mathbf{A}_{\mathrm{S}} \coloneqq \mathbf{A}_{\mathcal{F}^{\mathrm{i}}\mathcal{F}^{\mathrm{i}}} - \mathbf{A}_{\mathcal{F}^{\mathrm{i}}\mathcal{T}}\mathbf{A}^{-1}_{\mathcal{T}\mathcal{T}}\mathbf{A}_{\mathcal{T}\mathcal{F}^{\mathrm{i}}}. \tag{45}$$

3 Generic implementation tools for HHO methods

In this section, we describe generic programming tools for the implementation of DiSk methods. We first present in Subsection 3.1 some tools related to the mesh data structure and the operations to be performed on the elements of the mesh such as quadratures. These tools have a relatively wide scope, and can actually be used to implement other methods than DiSk ones. They are in essence similar to the tools used in other libraries using generic programming to different extents, as in [4, 6, 21, 27, 30, 31]. Then, in Subsection 3.2, we discuss generic programming tools that are more specific to DiSk methods. For simplicity, we focus on HHO methods as an illustrative example, but the presentation can be adapted to other DiSk methods. We provide several (short) listings to illustrate the generic implementation. For further insight, the reader may consult the library, named DiSk++, which is available as open-source under MPL License at the address `https://github.com/datafl4sh/diskpp`.

3.1 Mesh data structure

Although our main focus is on DiSk methods for which only mesh cells and faces are the relevant geometric objects, we slightly broaden the discussion here so as to include mesh edges and vertices as well. For a d-dimensional mesh, the cells are its elements of dimension d and the faces are its elements of dimension $(d-1)$. We consider specifically cells that are d-polytopes (not necessarily convex) and their faces are planar. Before getting started, let us provide a simple example of the level of generality we want to achieve. The following listing describes how to implement in the present setting a function to compute the measures of all the mesh elements.

```
template<typename Mesh>
void list_measures(const Mesh& msh)
{
    for (auto& cl : msh)
    {
        std::cout << measure(msh, cl) << std::endl;
        auto fcs = faces(msh, cl);

        for (auto& fc : fcs)
            std::cout << " - " << measure(msh, fc) << std::endl;
    }
}
```

Listing 1: The code of this listing is an example of the level of generality we want to achieve in our library. The code computes the measure of the mesh elements, and is suitable for any mesh.

3.1.1 Abstracting the various geometric entities

At the most basic level, a mesh is composed by different geometric entities having specific shapes: nodes, edges, triangles, quadrangles, tetrahedra, hexahedra and so on. Depending on the *kind* of the elements that a mesh contains, we can categorize the mesh as simplicial, hexahedral or - in

the most general cases - polyhedral. Keeping visible all the information about the exact kind of elements, however, would prevent us from achieving the desired level of generality. We therefore introduce three levels of abstraction:

- Transforming the mesh file in an internal representation of "shapes",
- Mapping the "shapes" to generic nodes, edges, surfaces and volumes (n-polytopes with $0 \leq n \leq d$),
- Mapping d-polytopes to cells, $(d-1)$-polytopes to faces, and so on.

Note that the role of the abstraction is to *hide* information, and not to discard it. This means that while writing code, the user does not care about what a cell or a face actually is, but the library at all times has the knowledge of all the details. Note also that we keep a representation for the various geometric mesh entities since in HHO methods for instance, mesh faces have to be handled independently of the mesh cells. Moreover, keeping a fully explicit mesh representation allows us to implement, in addition to DiSk methods, also discretization methods requiring the access to elements different than cells and faces (e.g., conforming finite elements).

First level of abstraction The first level of abstraction can be implemented in the *loaders* and in the *storage classes*. The role of a loader is to read a mesh file, find the "shapes" present in the mesh, and generate shape objects that have some representation (reflected in the code by specific *types*). That representation is dictated by the storage classes.

A storage class is a class tailored to encapsulate the representation of a specific kind of element (which we call *physical element*) as efficiently as possible, exploiting the features of that element to optimize memory usage and computational speed. Various storage classes can be provided, for instance, `simplicial_element`, the `hexahedral_element` and `generic_element`. Objects created from the `simplicial_element` and `hexahedral_element` storage classes are very lightweight and do not use dynamic memory, but they are limited in the kind of elements (and thus shapes) they can represent. On the other hand, objects created from the `generic_element` class can handle any kind of element, but they require the usage of dynamic memory and are more expensive. Some performance comparison between the different storage classes will be shown in Section 4 below.

To summarize, with the first abstraction level, the details of all the different mesh file formats are hidden because elements are stored inside objects belonging to specific storage classes.

Second level of abstraction The second level of abstraction introduces the concepts of nodes, edges, surfaces and volumes. We will call these objects *abstract mesh elements*. The goal of the second abstraction layer is to map the abstract elements to the physical elements. For example, in the case of a simplicial mesh, surfaces are mapped to triangles and volumes to tetrahedra.

This mapping is established in the *mesh storage*, via the *tag* `StorageClass` (see Listing 2), which is required to instantiate the storage class trait and to retrieve the correct types of the physical elements. In this way, from the user's point of view, a mesh storage is nothing more than a container for abstract elements; however, all the required information about the physical elements is retained in the proper way.

```
template<typename T, size_t DIM, typename StorageClass>
class mesh_storage /* Main template */
{
    static_assert(DIM > 0 && DIM <= 3, "Only 1D, 2D and 3D");
};

template<typename T, typename StorageClass>
struct mesh_storage<T, 3, StorageClass> /* 3D specialization*/
{
    typedef storage_class_trait<StorageClass, 3>    sc_trait;
    typedef typename sc_trait::volume_type          volume_type;
    typedef typename sc_trait::surface_type         surface_type;
    typedef typename sc_trait::edge_type            edge_type;
    typedef typename sc_trait::node_type            node_type;

    std::vector<volume_type>                        volumes;
    std::vector<surface_type>                       surfaces;
    std::vector<edge_type>                          edges;
    std::vector<node_type>                          nodes;
};
```

Listing 2: Definition of the mesh storage. The 3D specialization is shown; the 2D specialization does not have the `volumes` member, while the 1D one has only the members `nodes` and `edges`.

The advantage of this structure is that, when a new kind of mesh has to be supported, the modifications to the code are very localized. To add the support for a new kind of elements, it is just needed to write:

- The storage classes that encapsulate the implementation details of the new physical elements,
- The loader (or the loaders) for the new mesh format(s),
- The trait specialization that specifies which are the actual types of the new physical elements.

Third level of abstraction The third level of abstraction is obtained by a template called `mesh`, which (among other things) provides the methods to obtain the iterators on the cells and on the faces. The `mesh` template has three specializations, which correspond to the dimensions 1, 2 and 3; moreover, to be instantiated it needs also the type of the underlying storage. Thus, depending on the dimension, the methods of `mesh` that return the iterators on the cells and on the faces do nothing else than redirecting the call to the correct methods of its underlying storage. In other words, if cell iterators are requested on a 3D mesh, the call is redirected to the member `volumes` of the storage, while if face iterators of a 2D mesh are requested, the call is redirected to the member `nodes` of the underlying storage. For reasons of modularity and decoupling, this third level of abstraction is actually implemented by a hierarchy of three template classes: `mesh_bones`, `mesh_base` and `mesh` (see Figure 4). The user interacts only with `mesh`.

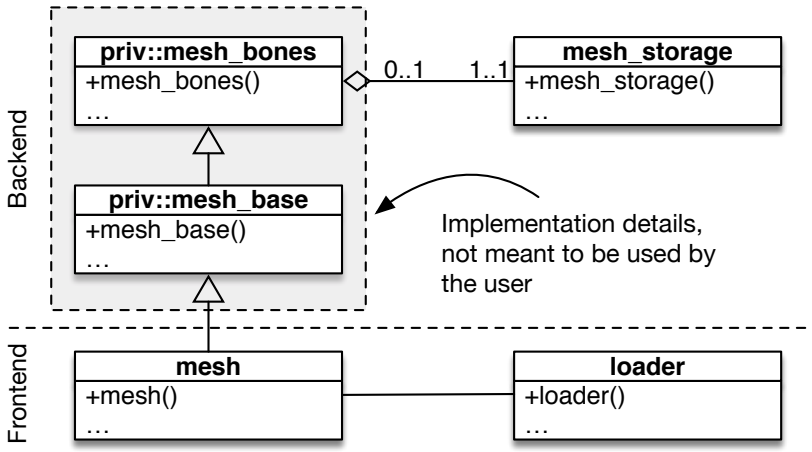

Figure 4: UML (Uniform Modeling Language) diagram of the class hierarchy implementing the mesh data structure.

We emphasize that all the classes defined above (the storage and the classes of the mesh hierarchy) are fully generic. A mesh becomes polyhedral, hexahedral, simplicial only because of the `StorageClass` discussed before. Moreover, all the abstractions depend exclusively on static (i.e., compile time) information: this means that the implementation avoids the costs of dynamic method selection or other mechanisms found in object-oriented programming.

3.1.2 Queries on the mesh elements

An essential capability of the above generic programming tools is to enable queries about properties of the elements, like barycenter, volume, area and so on. Since the main goal is to provide a uniform interface across different kinds of meshes, the functions computing these quantities are templates parametrized on the type of the mesh and the type of the element.

Geometry-related queries The `barycenter()` function for example (see Listing 3) is parametrized in a way that matches any kind of mesh and any kind of element, since in every case the barycenter is computed by adding (coordinate-wise) all the points and by dividing by their cardinality.

```
template<typename Mesh, typename Element>
typename Mesh::point_type
barycenter(const Mesh& msh, const Element& elm)
{
    auto pts = points(msh, elm);
    auto bar = accumulate(next(pts.begin()), pts.end(), pts.front());
    return bar / typename Mesh::coordinate_type( pts.size() );
}
```

Listing 3: Computation of the barycenter

The `measure()` function, however, has different versions with parametrizations that match only specific kinds of meshes and elements.

```
template<typename T>
T measure(const generic_mesh<T,2>& msh,
          const typename generic_mesh<T,2>::face& fc);
```

Listing 4: Prototype of the function that computes the measure for a 2D generic face

In Listing 4, the signature matches only 2D general meshes and in this case the function will compute and return a length. In Listing 5, however, the signature matches only the faces of a 3D simplicial mesh and then the corresponding function will compute and return an area.

```
template<typename T>
T measure(const simplicial_mesh<T, 3>& msh,
          const typename simplicial_mesh<T, 3>::face& surf);
```

Listing 5: Prototype of the function that computes the measure for a 3D simplicial face

Different functions will therefore be called on different kinds of mesh, and the right function to use is selected at compile time by the compiler. This selection is based exclusively on information available statically to the compiler, and therefore has *zero* runtime overhead. To conclude, when the support for a specific operation has to be added to a new mesh format, it suffices to add the right function specialization.

Polynomials, basis functions and quadratures Polynomial basis functions and quadrature rules can also be made available as generic templates working on any kind of mesh. For instance, if one wants to compute the integral $(p, q)_T$ on all the elements of the mesh, with $p, q \in \mathbb{P}_d^k(T)$, one just needs the code shown in Listing 6, which is mesh-independent.

```
template<typename Mesh>
void compute_integrals(const Mesh& msh)
{
    /* assumes that degree, p and q are correctly defined */
    typedef Mesh                                mesh_type;
    typedef typename mesh_type::cell_type   cell_type;

    scaled_monomial_scalar_basis<mesh_type, cell_type> cb(degree);
    quadrature<mesh_type, cell_type> cq(2*degree);
    for (auto& cl : msh)
    {
        auto quadpoints = cq.integrate(msh, cl);
        for (auto& qp : quadpoints)
        {
            auto phi  = cb.eval_functions(msh, cl, qp.point());
            mass_matrix += qp.weight() * phi * phi.transpose();
        }

        std::cout << "(p,q) = " << dot(q, mass_matrix*p) << std::endl;
    }
}
```

Listing 6: Example of a function to compute an integral

3.2 Generic HHO Implementation

HHO methods only use mesh cells as faces. The overview of the mesh data structure for HHO methods is schematized in Figure 5.

The two key ingredients in HHO methods are the gradient reconstruction operator (corresponding to ∇R_T^{k+1}) and the stabilization operator. To compute the gradient reconstruction operator, one can use the `gradient_reconstruction` template, that can be instantiated as in Listing 7.

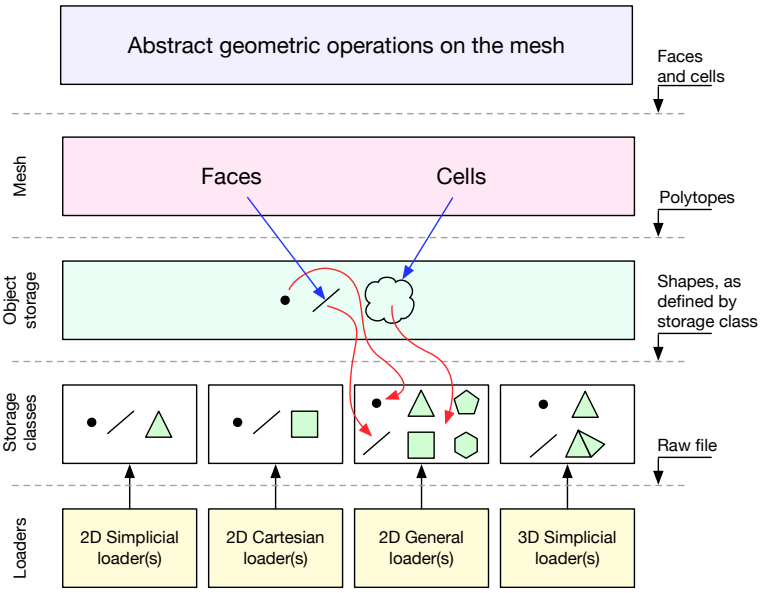

Figure 5: Overview of the mesh data structure for HHO.

```
gradient_reconstruction<mesh_type,
                        cell_basis_type,
                        cell_quadrature_type,
                        face_basis_type,
                        face_quadrature_type> gradrec(degree);
for (auto& cl : msh)
{
    gradrec.compute(msh, cl);
    // gradrec.oper contains the operator
    // gradrec.data contains a_T^(1)
}
```

Listing 7: Gradient reconstruction operator declaration. The template implementing gradient reconstruction must be instantiated specifying the type of the mesh, the types of the bases and the types of the quadratures. These parameters allow the compiler to specialize the operator to the specific kind of problem being solved. Then, the actual operator can be computed inside of the assembly loop by calling the method `compute()`, which is the counterpart of the pseudocode of Algorithm 1.

To compute the stabilization, one can use the `diffusion_like_stabilization` template, that can be instantiated as in Listing 8.

```
diffusion_like_stabilization<mesh_type,
                             cell_basis_type,
                             cell_quadrature_type,
                             face_basis_type,
                             face_quadrature_type> stab(degree);

for (auto& cl : msh)
{
    stab.compute(msh, cl);
    // stab.data contains a_T^(2)
}
```

Listing 8: Stabilization operator declaration. The stabilization operator, like the gradient reconstruction, must be instantiated specifying the type of the mesh, the types of the bases and the types of the quadratures.

There are similar template classes specific for the static condensation and for the assembler, that are omitted for brevity.

With all these classes instantiated, after loading the mesh with a specific *loader* that fills the `msh` object, the assembly phase of the problem reduces to the code described in Listing 9.

```
for (auto& cl : msh)
{
    /* build a_T^(1) + a_T^(2) */
    gradrec.compute(msh, cl);
    stab.compute(msh, cl, gradrec.oper);

    auto cell_rhs =
        disk::compute_rhs<cell_basis_type,
                          cell_quadrature_type>(msh, cl, load, degree);

    /* local cell contribution: a_T^(1) + a_T^(2) */
    matrix_type loc = gradrec.data + stab.data;

    /* do static condensation */
    auto sc = statcond.compute(msh, cl, loc, cell_rhs);
    assembler.assemble(msh, cl, sc);
}

assembler.impose_boundary_conditions(msh, bc_func);
assembler.finalize();
```

Listing 9: Assembly of the model diffusion problem using the HHO method.

The code can run on *any* mesh of the kinds supported by the library. Once the code above

has executed, the *assembler* object provides two members, `matrix` and `rhs`, that give access to the global linear system. These two members can then be passed to a suitable solver.

3.2.1 Choice of basis functions

As discussed in Section 2, we need polynomial basis functions associated with the cells and the faces of the mesh. In the results reported below, we used a basis composed of *scaled monomials* to generate them. Let $\bar{\mathbf{x}}_T \in \mathbb{R}^d$ be the barycenter of the cell T, $\mathbf{x} \in T$ a point in T, and h_T the diameter of T defined as the maximal distance between two vertices. A scaled monomial is of the form

$$m_T(\mathbf{x}) = \prod_{i=1}^{d} \tilde{x}_{T,i}^{\alpha_i}, \tag{46}$$

where $\tilde{\mathbf{x}}_T = (\mathbf{x} - \bar{\mathbf{x}}_T)/h_T$ and $\tilde{x}_{T,i}$ is the i-th component of $\tilde{\mathbf{x}}_T$. The scaled monomial basis for $\mathbb{P}_d^k(T)$ is formed by taking all the monomials $m_T(\mathbf{x})$ of degree up to k:

$$\mathbb{P}_d^k(T) = \operatorname{span}\left\{ \prod_{i=1}^{d} \tilde{x}_{T,i}^{\alpha_i} \mid 1 \le i \le d \ \wedge \ 0 \le \sum_{i=1}^{d} \alpha_i \le k \right\}. \tag{47}$$

One advantage of the above choice of basis functions is that the two quantities that depend on T ($\bar{\mathbf{x}}_T$ and h_T) can be coded in a generic fashion, independent of the actual shape of the element.

Regarding the basis functions associated with the faces, since they are $(d-1)$-variate polynomials, we need to consider the affine bijective mapping $\mathbf{T}_F : \mathbb{R}^{d-1} \to H_F$ introduced in Section 2.1.1. To this aim, we introduce a local coordinate system on the face F with origin at the barycenter $\mathbf{x}_F$ and with local coordinates denoted by $\boldsymbol{\xi}_F = (\xi_{F,i})_{1 \le i \le d-1}$. This coordinate system is obtained by choosing $(d-1)$ edges of F with a vertex in common and such that they give rise to a set of linearly independent edge vectors $(\mathbf{k}_i)_{1 \le i \le d-1}$. These vectors are subsequently made orthonormal using the Gram–Schmidt process as originally proposed in [5] in the context of Discontinuous Galerkin methods. Given a point $\mathbf{x} \in F$, its local coordinates $(\xi_{F,i})_{1 \le i \le d-1}$ are computed by first calculating the rescaled vector $\tilde{\mathbf{x}}_F = (\mathbf{x} - \bar{\mathbf{x}}_F)/h_F$ and then by projecting $\tilde{\mathbf{x}}_F$ on all the vectors $\mathbf{k}_i$. A scaled monomial is of the form

$$m_F(\boldsymbol{\xi}) = \prod_{i=1}^{d-1} \xi_{F,i}^{\alpha_i}. \tag{48}$$

The scaled monomial basis for $\mathbb{P}_{d-1}^k(F)$ is finally formed by taking all the monomials $m_F(\boldsymbol{\xi})$ of degree up to k:

$$\mathbb{P}_{d-1}^k(F) = \operatorname{span}\left\{ \prod_{i=1}^{d-1} \xi_{F,i}^{\alpha_i} \mid 1 \le i \le d-1 \ \wedge \ 0 \le \sum_{i=1}^{d-1} \alpha_i \le k \right\}. \tag{49}$$

3.2.2 Quadratures

The code employs different quadratures, depending on the kind of element on which integration is required. On edges, standard Gauss quadrature is used. When integration on triangles and on tetrahedra is needed, the Dunavant quadrature [23] and the Grundmann–Moeller quadrature [26] are used respectively. If elements are quadrilaterals or hexahedra, the quadrature is obtained by tensorizing the one-dimensional Gauss quadratures. Finally, when the elements are polytopes that

are star-shaped with respect to their barycenter (as is the case for the meshes we consider), they are broken up into simplices and the integration is computed simplex by simplex. The appropriate quadrature is selected statically during the instantiation of the `quadrature` template. The efficiency of generic quadratures on elements for which specific quadratures are available (e.g., simplices) is guaranteed by a templated implementation resembling the pseudocode provided in Algorithm 2.

Algorithm 2 Selection of appropriate quadratures. According to the data type of the cell, the compiler selects at compile time the appropriate quadrature method. The user however sees only one `integrate()` function. $TetrahedralQuadratures(T)$ and $LineGaussPoints()$ are pseudocode functions returning sets of pairs of quadrature points and corresponding weights, as in Algorithm 1.

```
1: template<typename CellType>
2: function INTEGRATE(CellType T)
3:     Q ← ∅
4:     {S_1, ..., S_n} ← SplitInSimplices(T)
5:     for S ∈ {S_1, ..., S_n} do
6:         Q_s ← TetrahedralQuadrature(S)
7:         Q ← Q ∪ Q_s
       return Q

8: template<>
9: function INTEGRATE(TetrahedralCellType T)
10:    Q ← TetrahedralQuadrature(T)
        return Q

11: template<>
12: function INTEGRATE(CartesianCellType T)
13:    G ← LineGaussPoints()
14:    Q ← TensorizeGaussPoints(T, G)
        return Q
```

4 Profiling on a model elliptic problem

The goal of this section is to present a profiling of the generic programming tools described in Section 3. We profiled different parts of the library, in particular the computation of the reconstruction operator, the computation of the stabilization operator, the static condensation and the solver. DiSk++ uses the Eigen library for the linear algebra operations, at the time of writing Eigen 3.3.1 was available. For the solution of the linear system, we used the PARDISO sparse linear solver from the Intel MKL library. The reported timings represent the total time, t_{cpu}, spent by the program on *all* the CPUs. Since the program was run on a 4-core CPU (Intel Core i7-3615QM), actual wall-clock times were much lower. We note that the assembly is sequential whereas the solver is run in parallel on 4 cores. The reported CPU times are as per `getrusage()`. The code has been compiled both with `clang` and `gcc`, and the usual optimization level is `-O3`.

We have collected data about the running times on 2D triangular and hexagonal meshes and on 3D tetrahedral, hexahedral, and polyhedral meshes. The 3D meshes were obtained from the

FVCA6 benchmark [25]; some meshes are illustrated in Figure 6. The timings are reported against the total number of face-based components in the linear system, which we denote by `DOFs`, i.e.,

$$\texttt{DOFs} := N^k_{\mathcal{F}^\mathrm{i}} = N_{\mathcal{F}^\mathrm{i}} \times N^k_{d-1}. \tag{50}$$

The right-hand term we used to solve (1) was $f = \sin(\pi x)\sin(\pi y)$ in the 2D case and $f = \sin(\pi x)\sin(\pi y)\sin(\pi z)$ in the 3D case, and the domain Ω was, respectively, the unit square and the unit cube.

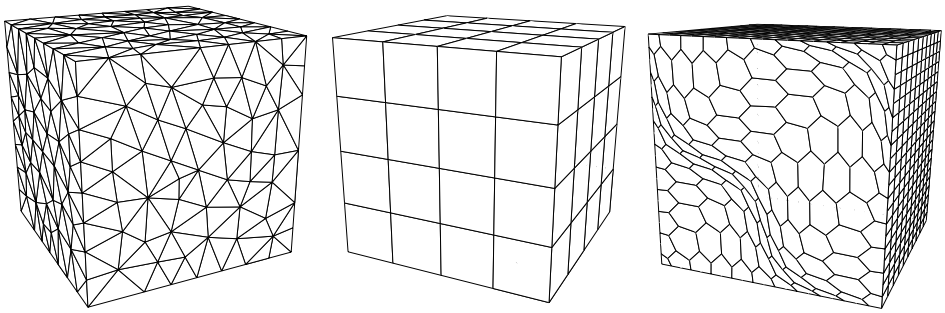

Figure 6: Examples of tetrahedral, hexahedral, and polyhedral meshes from the FVCA6 benchmark used in the tests.

4.1 2D test cases

The 2D test cases were run on triangular and hexagonal meshes. The triangular meshes have 56, 224, 896, and 3584 elements respectively, whereas the hexagonal meshes have 76, 280, 1072, and 4192 elements respectively. In Figures 7, 8, and 9, we compare the running time of the various parts of the computation process, in particular the assembly (Reconstruction operator, Stabilization operator, Static condensation) and the global linear system solution (Solver). The computation times are obtained on meshes of triangles, a mix of triangles and hexagons and only hexagons respectively. The mixed meshes are obtained from the hexagonal meshes by splitting in triangles half of the hexagons. It is possible to see that the shape of the elements influences the computation time, this is in particular due to the quadratures that, to be computed on hexagons, must split the element in a collection of triangles. For all meshes, the assembly time scales linearly with the number of degrees of freedom. We observe that, on the coarser meshes, the running time for the linear solver may not be representative of the trend for finer meshes. Finally, in Table 1, we summarize the speedups obtained using specialized versus generic data structure on triangular meshes.

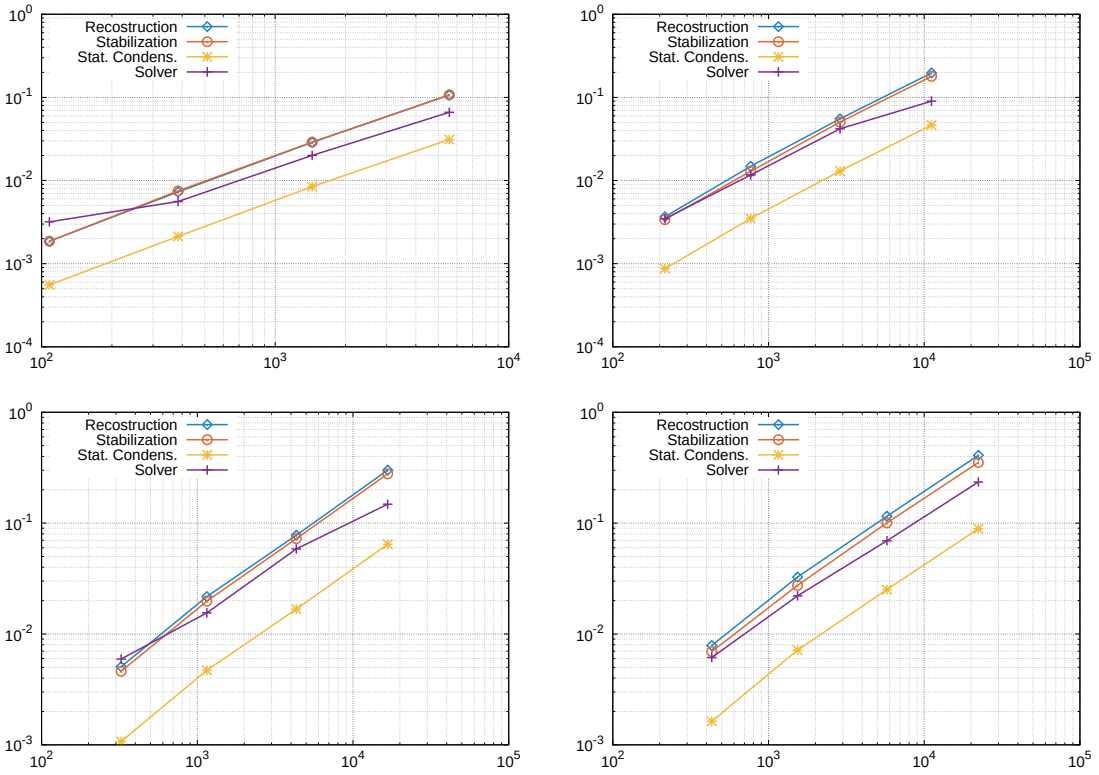

Figure 7: Timings on triangular meshes with respect to `DOFs` using generic data structure (from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$); (violet) crosses: Solver, (red) circles: Stabilization, (blue) diamonds: Reconstruction, (yellow) stars: Static condensation.

`DOFs`	Rec	Stab
108	1.25x	0.91x
384	1.45x	1.10x
1440	1.52x	1.24x
5568	1.83x	1.49x

`DOFs`	Rec	Stab
216	1.56x	1.33x
768	1.48x	1.18x
2880	1.78x	1.47x
11136	1.77x	1.47x

`DOFs`	Rec	Stab
324	1.83x	1.63x
1152	1.66x	1.44x
4320	1.53x	1.39x
16704	1.53x	1.39x

`DOFs`	Rec	Stab
432	1.59x	1.61x
1536	1.45x	1.35x
5760	1.35x	1.31x
22272	1.32x	1.27x

Table 1: Speedup in the assembly process (Reconstruction and Stabilization) due to the usage of a specialized data structure. Triangular meshes, from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$.

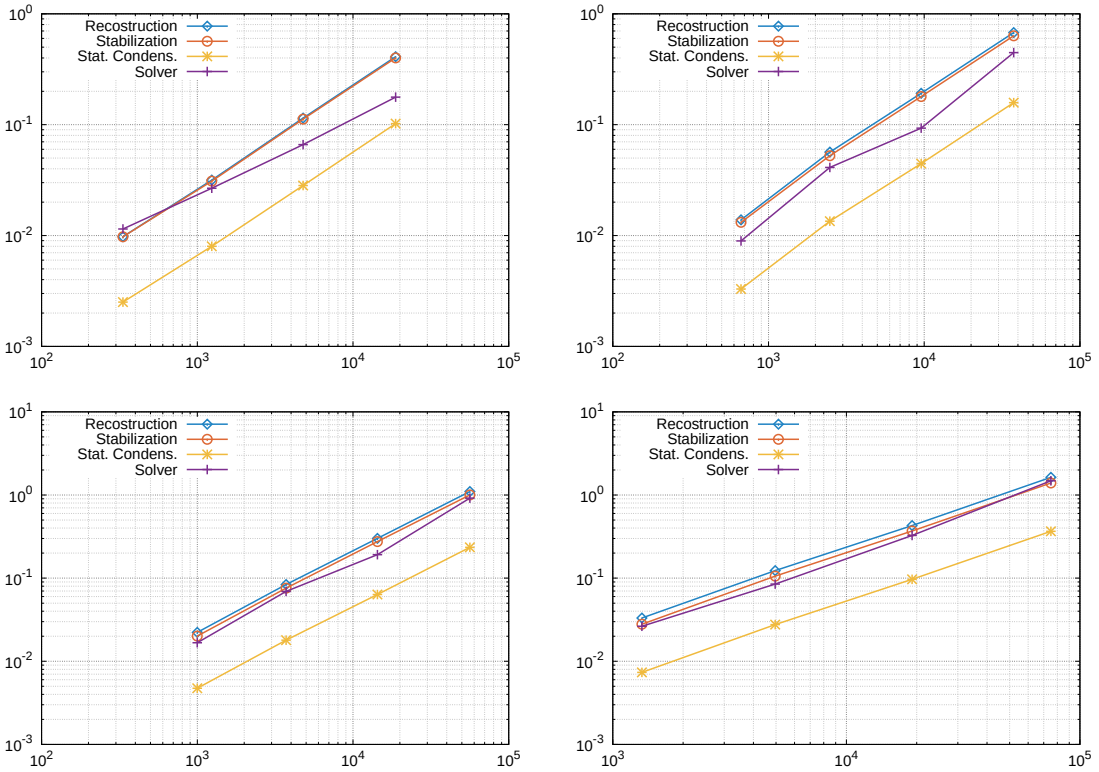

Figure 8: Timings on mixed triangular (50%) and hexagonal (50%) meshes with respect to `DOFs` using generic data structure (from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$); (violet) crosses: Solver, (red) circles: Stabilization, (blue) diamonds: Reconstruction, (yellow) stars: Static condensation.

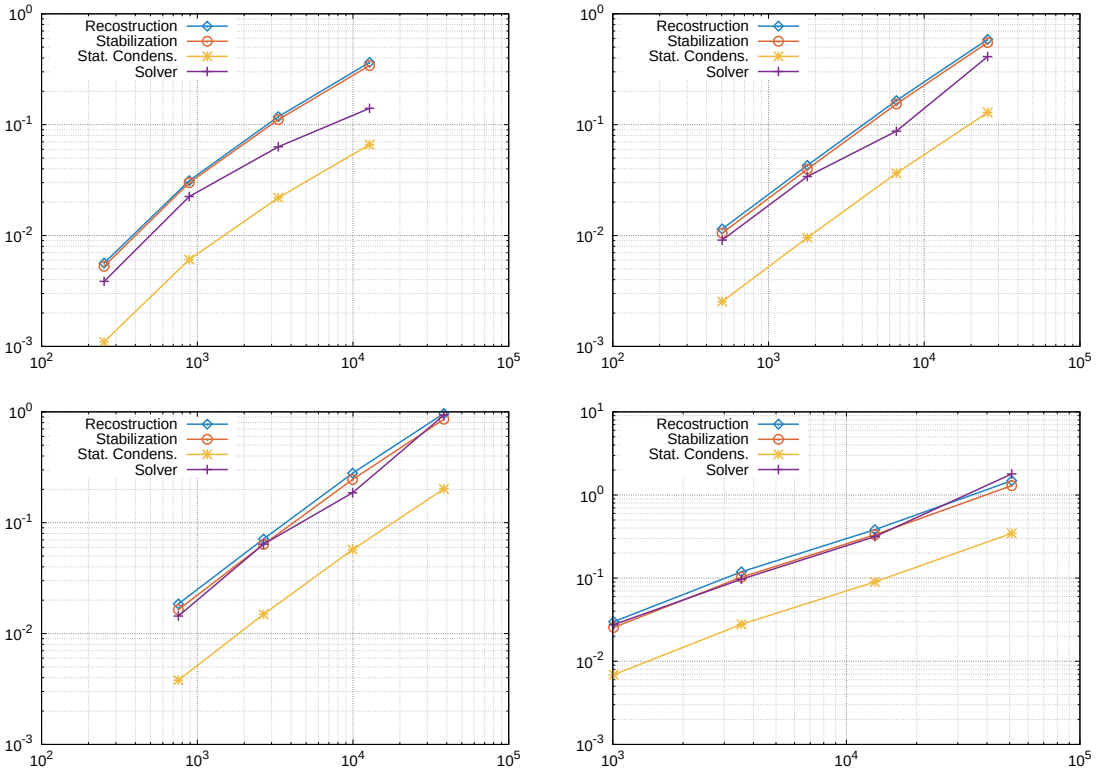

Figure 9: Timings on hexagonal meshes with respect to `DOFs` using generic data structure (from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$); (violet) crosses: Solver, (red) circles: Stabilization, (blue) diamonds: Reconstruction, (yellow) stars: Static condensation.

4.2 3D test cases

The 3D test cases were run on tetrahedral, hexahedral, and polyhedral meshes obtained from the FVCA6 benchmark set. The tetrahedral meshes have 2003, 3898, 7711 and 15266 elements, the hexahedral meshes have 8, 64, 512 and 4096 elements, and the polyhedral meshes have 1042, 8820 and 28830 elements. In Figures 10, 11, 12, 13 we compare the running time of the various parts of the computation process, in particular the assembly and the global linear system solution. The assembly time scales linearly with the number of degrees of freedom, as expected. In Tables 2 and 3, we summarize the speedups obtained using specialized versus generic data structure on tetrahedral and hexahedral meshes. We observe both in 2D and 3D that assembly times on specialized data structures are significantly lower than assembly times on the generic data structure, confirming that the approach taken in DiSk++ is advantageous. The speedup is particularly welcome in 3D, where the assembly process takes a significant part of the computation time. As a final remark, the solver was not run on the polyhedral meshes because of memory constraints.

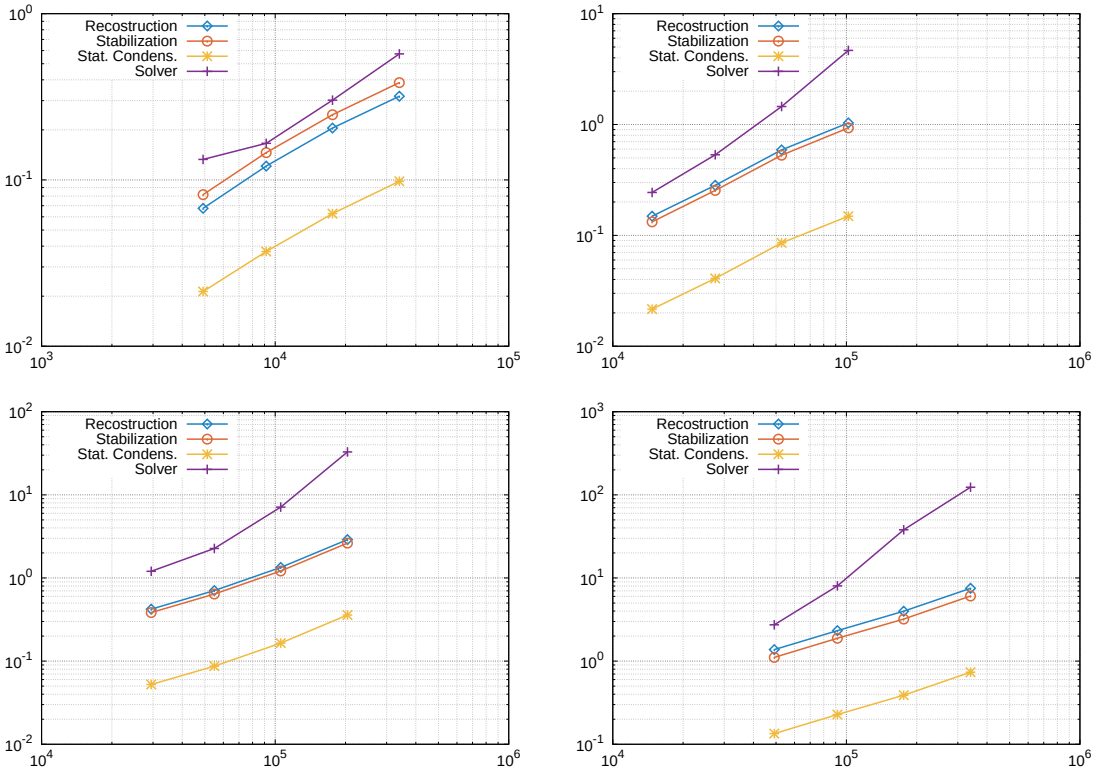

Figure 10: Timings on tetrahedral meshes with respect to `DOFs` using specialized data structure (from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$); (violet) crosses: Solver, (red) circles: Stabilization, (blue) diamonds: Reconstruction, (yellow) stars: Static condensation.

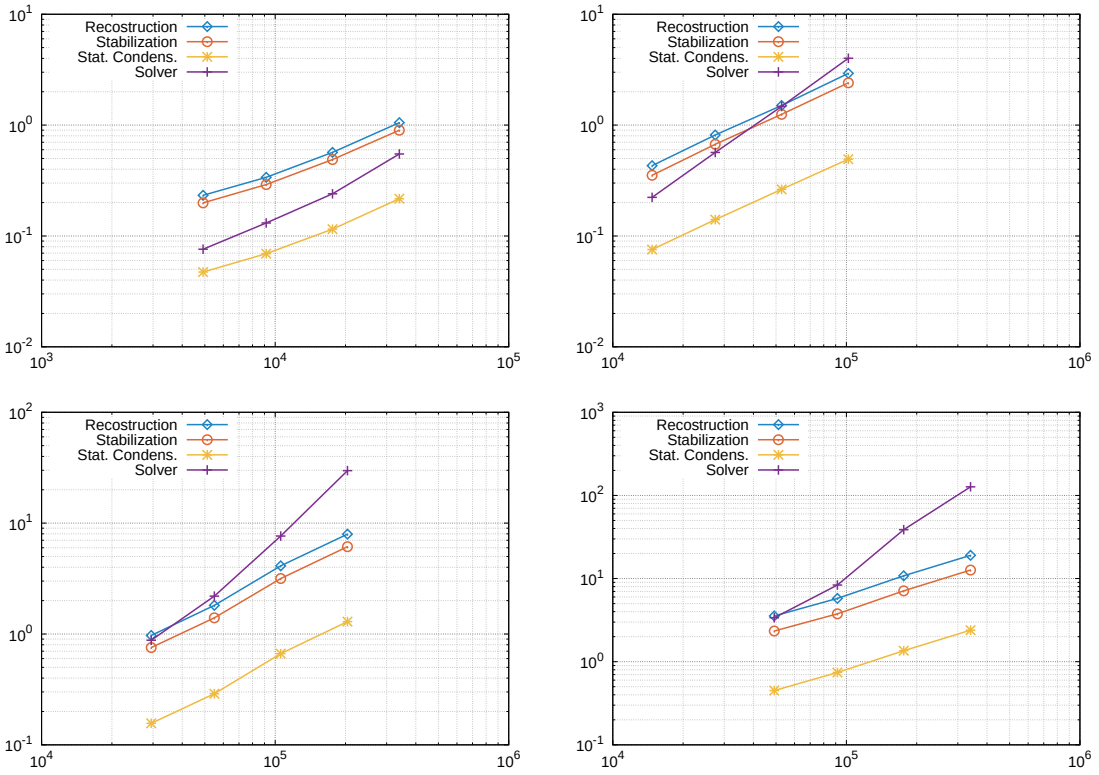

Figure 11: Timings on tetrahedral meshes with respect to `DOFs` using generic data structure (from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$); (violet) crosses: Solver, (red) circles: Stabilization, (blue) diamonds: Reconstruction, (yellow) stars: Static condensation.

`DOFs`	Rec	Stab
4912	3.45x	2.44x
9152	2.79x	1.99x
17600	2.77x	1.98x
34009	3.30x	2.34x

`DOFs`	Rec	Stab
14736	2.89x	2.66x
27456	2.88x	2.64x
52800	2.54x	2.36x
102027	2.83x	2.58x

`DOFs`	Rec	Stab
29472	2.30x	1.97x
54912	2.57x	2.18x
105600	3.08x	2.61x
204054	2.75x	2.33x

`DOFs`	Rec	Stab
49120	2.58x	2.11x
91520	2.48x	2.00x
176000	2.71x	2.22x
340090	2.53x	2.08x

Table 2: Speedup in the assembly process (Reconstruction and Stabilization) due to the usage of a specialized data structure. Tetrahedral meshes, from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$.

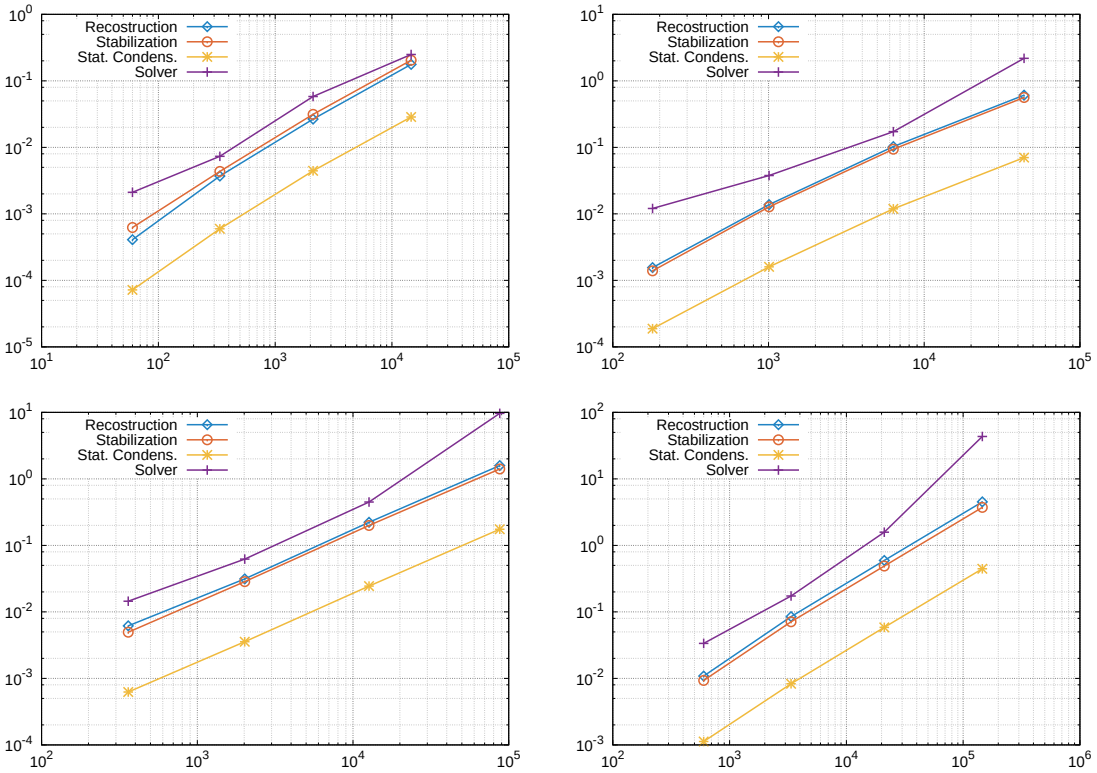

Figure 12: Timings on hexahedral meshes with respect to `DOFs` using specialized data structure (from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$); (violet) crosses: Solver, (red) circles: Stabilization, (blue) diamonds: Reconstruction, (yellow) stars: Static condensation.

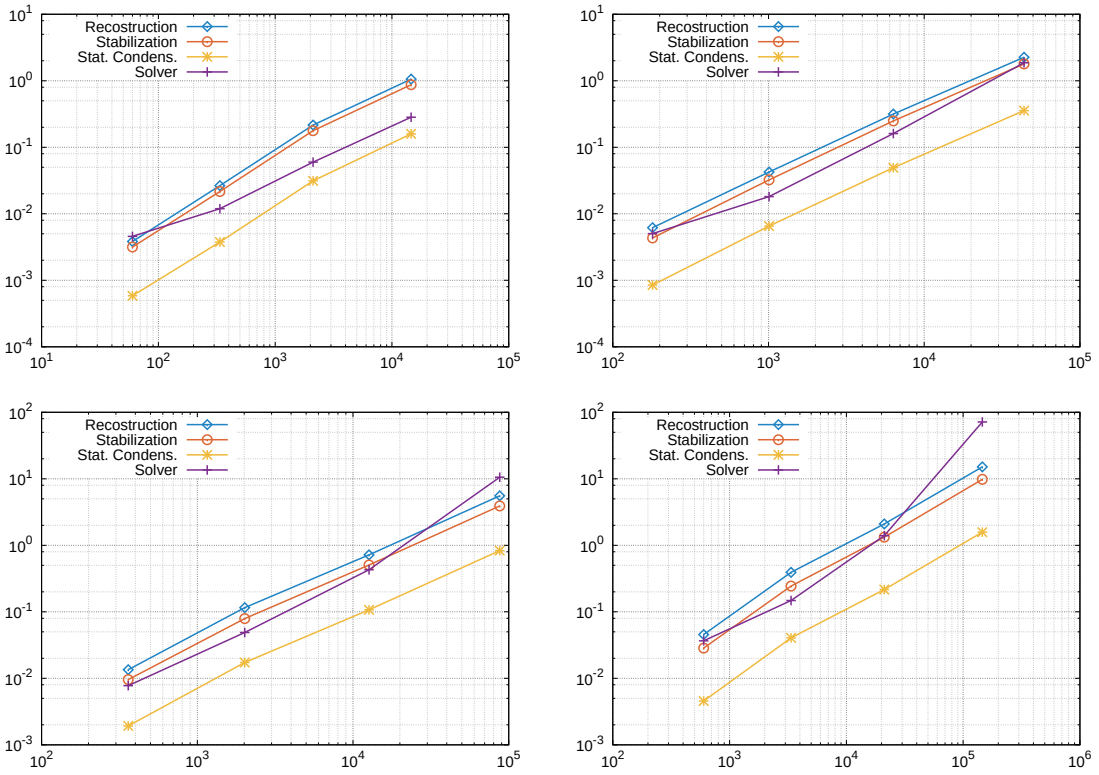

Figure 13: Timings on hexahedral meshes with respect to `DOFs` using generic data structure (from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$); (violet) crosses: Solver, (red) circles: Stabilization, (blue) diamonds: Reconstruction, (yellow) stars: Static condensation.

`DOFs`	Rec	Stab
60	9.37x	5.09x
336	7.15x	4.98x
2112	8.10x	5.67x
14592	5.98x	4.27x

`DOFs`	Rec	Stab
180	3.98x	3.11x
1008	3.08x	2.54x
6336	3.07x	2.66x
43776	3.70x	3.19x

`DOFs`	Rec	Stab
360	2.19x	1.94x
2016	3.69x	2.77x
12672	3.22x	2.54x
87552	3.50x	2.77x

`DOFs`	Rec	Stab
600	4.19x	3.06x
3360	4.62x	3.43x
21120	3.53x	2.71x
145920	3.35x	2.63x

Table 3: Speedup in the assembly process (Reconstruction and Stabilization) due to the usage of a specialized data structure. Hexahedral meshes, from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$.

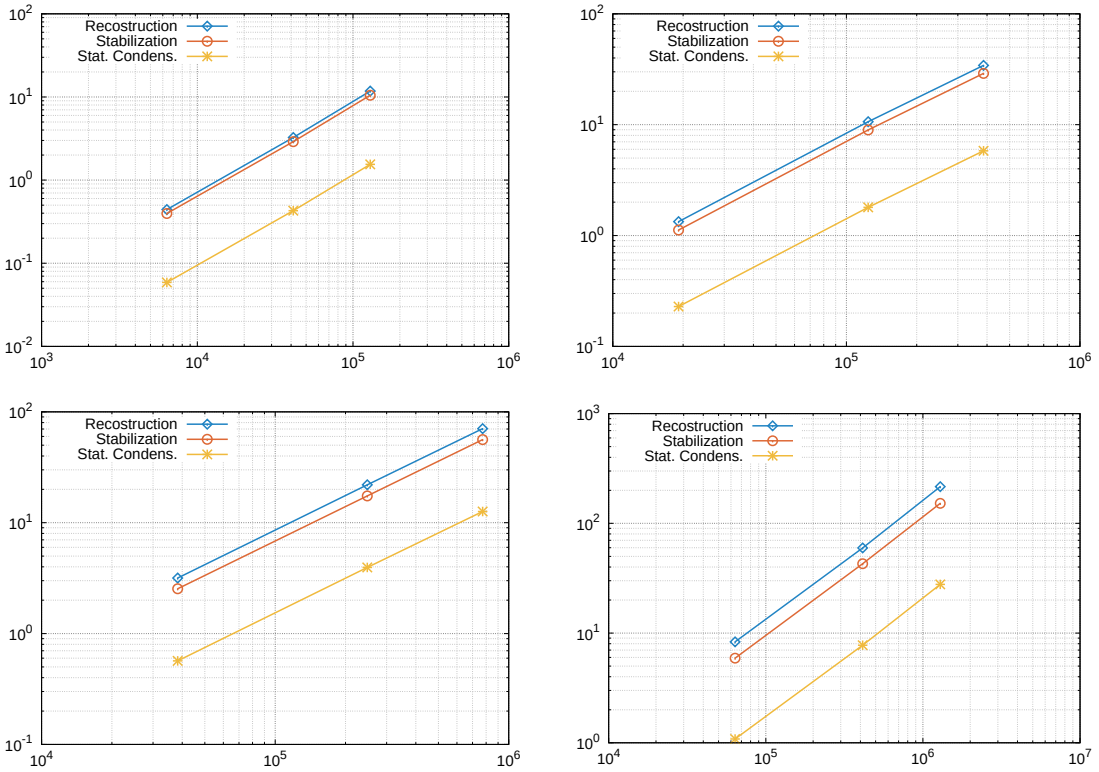

Figure 14: Timings on polyhedral meshes with respect to `DOFs` using generic data structure (from left to right and from top to bottom: $k = 0$, $k = 1$, $k = 2$, $k = 3$); (red) circles: Stabilization, (blue) diamonds: Reconstruction, (yellow) stars: Static condensation.

4.3 Analysis of computational effort vs. accuracy

One of the most important advantages offered by arbitrary-order methods is that for a fixed computational effort, higher polynomial orders yield better approximations of the problem solution. We analyzed this aspect in the HHO method, and we summarized our findings in Table 4. For this analysis, we considered a sequence of 4 triangular meshes, the first three ones are from the previously considered sequence in Subsection 4.1 (we dropped the first one which is too coarse for a reliable operation count), and we added a further refined mesh. On those meshes, we solved problem (1) and we measured the number of floating point operations required by the solver. This number (which is somewhat solver-dependent) was obtained direcly by querying the linear solver, which in our case is the PARDISO solver from the Intel MKL package. As expected, higher-order polynomial degrees outperform lower orders; for example, using $k = 3$ on the coarsest mesh gives an L^2-norm error which is two orders of magnitude smaller than using $k = 0$ on the finest mesh, at one tenth of the computational expense measured in Mflops.

	224 elems			896 elems			3584 elems			14336 elems		
k	F	L^2-err	DOFs	F	L^2-err	DOFs	F	L^2-err	DOFs	F	L^2-err	DOFs
0	0	1.72e-2	384	0	4.29e-3	1440	2	1.07e-2	5568	23	2.68e-4	21888
1	0	8.07e-4	768	2	1.01e-4	2880	21	1.26e-5	11136	169	1.57e-6	43776
2	1	3.38e-5	1152	8	2.12e-6	4320	73	1.33e-7	16704	582	8.30e-9	65664
3	2	1.06e-6	1536	20	3.31e-8	5760	173	1.04e-9	22272	1355	3.24e-11	87552

Table 4: Computational effort (column F, in Mflops) vs. accuracy (error measured in L^2-norm) on various meshes and for various polynomial orders k.

5 Conclusions

In this work, we have presented generic programming tools to implement the recently-devised HHO methods, and we have profiled the implementation on a model elliptic problem in two and three space dimensions. As an interesting follow-up of this work, it would be interesting to use similar generic programming tools to implement, e.g., HDG methods and to profile the implementation on the above test cases. Profiling on more challenging model problems would also be very valuable.

Acknowledgment. This work was partially supported by the project HHOMM (ANR-15-CE40-0005).

References

[1] J. Aghili, S. Boyaval, and D. A. Di Pietro. Hybridization of mixed high-order methods on general meshes and application to the Stokes equations. *Comput. Methods Appl. Math.*, 15(2):111–134, 2015.

[2] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.

[3] B. Ayuso de Dios, K. Lipnikov, and G. Manzini. The nonconforming virtual element method. *ESAIM Math. Model. Numer. Anal.*, 50(3):879–904, 2016.

[4] W. Bangerth, D. Davydov, T. Heister, L. Heltai, G. Kanschat, M. Kronbichler, M. Maier, B. Turcksin, and D. Wells. The `deal.II` library, version 8.4. *Journal of Numerical Mathematics*, 24, 2016.

[5] 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(1):45–65, 2012.

[6] P. Bastian, F. Heimann, and S. Marnach. Generic implementation of finite element methods in the distributed and unified numerics environment (DUNE). *Kybernetika (Prague)*, 46(2):294–315, 2010.

[7] L. Beirão da Veiga, K. Lipnikov, and G. Manzini. *The Mimetic Finite Difference Method for Elliptic Problems*, volume 11 of *MS&A*. Springer, New York, 2014.

[8] D. Boffi, M. Botti, and D. A. Di Pietro. A nonconforming high-order method for the Biot problem on general meshes. *SIAM J. Sci. Comput.*, 38(3):A1508–A1537, 2016.

[9] F. Brezzi, K. Lipnikov, and M. Shashkov. Convergence of the mimetic finite difference method for diffusion problems on polyhedral meshes. *SIAM J. Numer. Anal.*, 43(5):1872–1896, 2005.

[10] F. Chave, D. A. Di Pietro, F. Marche, and F. Pigeonneau. A hybrid high-order method for the Cahn-Hilliard problem in mixed form. *SIAM J. Numer. Anal.*, 54(3):1873–1898, 2016.

[11] B. Cockburn. Static condensation, hybridization, and the devising of the HDG methods. In G. R. Barrenechea, F. Brezzi, A. Cangiani, and E. H. Georgoulis, editors, *Building Bridges: Connections and Challenges in Modern Approaches to Numerical Partial Differential Equations*, volume 114 of *Lecture Notes in Computational Science and Engineering*, pages 129–178, Switzerland, 2016. Springer.

[12] B. Cockburn, D. A. Di Pietro, and A. Ern. Bridging the Hybrid High-Order and Hybridizable Discontinuous Galerkin methods. *ESAIM: Math. Model Numer. Anal. (M2AN)*, 50(3):635–650, 2016.

[13] B. Cockburn, G. Fu, and F. J. Sayas. Superconvergence by M-decompositions. Part I: General theory of HDG methods for diffusion. *Math. Comp.*, 2016. To appear.

[14] B. Cockburn, J. Gopalakrishnan, and R. Lazarov. Unified hybridization of discontinuous Galerkin, mixed, and continuous Galerkin methods for second order elliptic problems. *SIAM J. Numer. Anal.*, 47(2):1319–1365, 2009.

[15] D. A. Di Pietro, J. Droniou, and A. Ern. A discontinuous-skeletal method for advection-diffusion-reaction on general meshes. *SIAM J. Numer. Anal.*, 53(5):2135–2157, 2015.

[16] D. A. Di Pietro and A. Ern. A Hybrid High-Order locking-free method for linear elasticity on general meshes. *Comput. Meth. Appl. Mech. Engrg.*, 283:1–21, 2015.

[17] D. A. Di Pietro and A. Ern. Arbitrary-order mixed methods for heterogeneous anisotropic diffusion on general meshes. *IMA J. Numer. Anal.*, 37(1):40–63, 2017.

[18] D. A. Di Pietro, A. Ern, and S. Lemaire. An arbitrary-order and compact-stencil discretization of diffusion on general meshes based on local reconstruction operators. *Comput. Meth. Appl. Math.*, 14(4):461–472, 2014.

[19] D. A. Di Pietro, A. Ern, and S. Lemaire. A review of Hybrid High-Order methods: formulations, computational aspects, comparison with other methods. In G. R. Barrenechea, F. Brezzi, A. Cangiani, and E. H. Georgoulis, editors, *Building Bridges: Connections and Challenges in Modern Approaches to Numerical Partial Differential Equations*, volume 114 of *Lecture Notes in Computational Science and Engineering*, pages 205–236, Switzerland, 2016. Springer.

[20] D. A. Di Pietro, A. Ern, A. Linke, and F. Schieweck. A discontinuous skeletal method for the viscosity-dependent Stokes problem. *Comput. Methods Appl. Mech. Engrg.*, 306:175–195, 2016.

[21] D. A. Di Pietro, J.-M. Gratien, and C. Prud'homme. A domain-specific embedded language in C++ for lowest-order discretizations of diffusive problems on general meshes. *BIT*, 53(1):111–152, 2013.

[22] 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 Mathematical Models and Methods in Applied Sciences*, 20(2):1–31, 2010.

[23] D. Dunavant. High degree efficient symmetrical gaussian quadrature rules for the triangle. *Int. J. Numer. Methods Engrg.*, 21:1129–1148, 1985.

[24] R. Eymard, T. 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(4):1009–1043, 2010.

[25] R. Eymard, G. Henry, R. Herbin, F. Hubert, R. Klofkorn, and G. Manzini. 3D benchmark on discretization schemes for anisotropic diffusion problems on general grids. In *Proceedings of Finite Volumes for Complex Applications VI*, pages 895–930. Springer, 2011.

[26] A. Grundmann and M. Moeller. Invariant integration formulas for the n-simplex by combinatorial methods. *SIAM Journal on Numerical Analysis*, 15(2):282–290, 1978.

[27] F. Hecht. New development in freefem++. *J. Numer. Math.*, 20(3-4):251–265, 2012.

[28] Y. Jeon and E.-J. Park. A hybrid discontinuous Galerkin method for elliptic problems. *SIAM J. Numer. Anal.*, 48(5):1968–1983, 2010.

[29] Y. Jeon, E.-J. Park, and D. Sheen. A cell boundary element method for elliptic problems. *Numer. Methods Partial Differential Equations*, 21(3):496–511, 2005.

[30] C. Prud'Homme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, C. Pena, C. Daversin, and C. Trophime. Advances in Feel++: a domain specific embedded language in C++ for partial differential equations. In *Eccomas12 - European Congress on Computational Methods in Applied Sciences and Engineering. Vienna, Austria*, 2012.

[31] P. Saramito. *Efficient C++ finite element computing with Rheolef: volume 2: discontinuous Galerkin methods*. CNRS and LJK, 2015. `http://cel.archives-ouvertes.fr/cel-00863021`.

[32] J. Wang and X. Ye. A weak Galerkin element method for second-order elliptic problems. *J. Comput. Appl. Math.*, 241:103–115, 2013.