A generic interface for parallel and adaptive scientific computing: Abstraction principles and the DUNE-FEM module

Andreas Dedner[1], Robert Klöfkorn[1], Martin Nolte[1], Mario Ohlberger[2]

[1]Abteilung für Angewandte Mathematik, Universität Freiburg, Germany
Email: dedner | nolte | robertk@mathematik.uni-freiburg.de

[2]Institut für Numerische und Angewandte Mathematik, Universität Münster, Germany
Email: mario.ohlberger@uni-muenster.de

Abstract

Starting from an abstract mathematical notion of discrete function spaces and operators, we derive a general abstraction for a large class of grid-based discretization schemes for stationary and instationary partial differential equations. Special emphasize is put on concepts for local adaptivity and parallelization with load balancing. The concepts are based on a corresponding abstract definition of a parallel and hierarchical adaptive grid given in [P. Bastian et al., Computing 82 (2008), no. 2-3, 103–119]. Unlike previous approaches, where the implementation of numerical schemes is based on particular abstractions for arrays and matrices, we describe an object oriented implementation of our abstraction in the DUNE-FEM library [http://dune.mathematik.uni-freiburg.de]. The leading design principle is a one-to-one correspondence between the mathematical objects and C++ interface classes. By using interface classes we manage to separate functionality from data structures. Thus, user implementations become independent of the underlying array or matrix implementations and the reorganization of data due to grid modification can be handled by the DUNE-FEM module. Efficiency is obtained by using modern template based generic programming techniques, including static polymorphism, the engine concept, and expression templates. We present numerical results for several benchmark problems and some advanced applications using the library DUNE-FEM. The experiments demonstrate both, the efficiency of the implementation and the applicability for a very large class of discretization schemes and applications.

AMS Subject Classifications: 65N30, 65Y05, 68U20

Key words: DUNE, finite elements, finite volumes, local discontinuous Galerkin, software, abstract interface, generic programming, C++, parallelization, adaptive methods, load balancing

Contents

1 Introduction

Starting from the eighties, there have continuously been attempts to develop general libraries that provide infrastructure for implementing discretization schemes for partial differential equations for increasing classes of applications. Most of these developments, however, were based on fixed data structures. This lead either to restricted applicability (cf. ALBERTA [33] - only simplex grids with bisection and finite elements), or to very huge packages that became more and more difficult to use (cf. UG [4] - all kind of elements, finite elements and finite volumes, adaptivity and parallelization, etc.). In recent years, there have been several attempts to overcome

such restrictions with the help of object oriented class libraries (cf. [40]). The benefit of such approaches is the possibility to separate functionality from specific data structures. Thus, the user can implement numerical schemes in a unified way, independent of the space dimension, the actual data structures of the underlying computational grid, or the particular array or matrix data structures used in the linear algebra.
In this contribution we are going to discuss such an approach for the abstraction of a very general class of grid-based discretization schemes for partial differential equations, including finite element, finite volume, and finite difference methods. The concept is based on an abstraction for general parallel and adaptive computational grids that was given in [6] and the corresponding implementation in the Distributed and Unified Numerics Environment Dune [5]. Our starting point is the abstract mathematical notion of discrete function spaces and operators that form the basis of a general abstraction for a large class of discretization schemes for stationary and instationary partial differential equations of any type. Special emphasis is put on concepts for local adaptivity and parallelization with dynamic load balancing. Unlike in most previous approaches, where the implementation of numerical schemes is based on particular abstractions for arrays and matrices (cf. [30] and the references therein), we realized an object oriented implementation of our abstraction in the Dune-Fem library (cf. [35]), using interface classes for discrete functions and operators. Our leading design principle is a one-to-one correspondence between objects in the abstraction on the one hand and the C++ interface classes on the other. Efficiency of the implementation is obtained by using modern template based generic programming techniques including static polymorphism, the engine concept, and expression templates.
The rest of the paper is organized as follows. In Subsection 1.1 we sketch the mathematical background for variational problems and corresponding discretization schemes. In Section 2 we then give an abstract description for grid-based discretization schemes for partial differential equations. This is done by first reflecting the abstract definition of a computational grid from [6], followed by the introduction of abstractions for discrete functions and operators. As our particular interest lies in the support of parallel and adaptive discretization schemes, we extend the abstract concept to handle such cases in Section 3. Example code snippets in Section 6 demonstrate how the abstract concepts are realized in the Dune-Fem module. Following the abstract definitions from Section 2, we describe in Section 4 a realization of a corresponding object-oriented class concept within the software library Dune-Fem. Finally, in Section 7 we present numerical experiments for several benchmark problems and some advanced applications. The range of schemes and applications prove the efficient usability of the presented design principles.

1.1 Mathematical background for variational problems

In order to derive an abstract description of discretization schemes for partial differential equations, let us first look at stationary problems of the form

$$L(v) = f$$

where $L : V \to W'$ is an operator mapping functions $v \in V$ into the dual space W' of some function space W and $f \in W'$ is a suitable right hand side. The above equality is then defined by the action of the functionals on functions $w \in W$, i.e.,

$$\langle L(v), w \rangle = \langle f, w \rangle \qquad \forall w \in W, \tag{1}$$

where $\langle \cdot, \cdot \rangle : W' \times W \to \mathbb{R}$ denotes the dual pairing between W' and W.
In practice, L will be the weak form of a differential operator and f will model forcing terms and, possibly, boundary conditions. A simple example is Poisson's equation, $-\Delta v = f$, in some domain Ω with a Dirichlet boundary condition $v = 0$ on $\partial\Omega$. In this case, $V = W = H_0^1(\Omega)$ and the weak solution is defined by (1) with

$$\langle L(v), w \rangle := \int_\Omega \nabla v \cdot \nabla w, \qquad \langle f, w \rangle := \int_\Omega f w.$$

Starting from this abstract definition of stationary problems, a large class of discretization schemes can be written in the abstract form

$$\langle L_h(v_h), w_h\rangle = \langle f_h, w_h\rangle \qquad \forall w_h \in W_h, \tag{2}$$

where now $L_h : V_h \to W_h'$ is a discrete operator (i.e., it acts on finite-dimensional function spaces V_h and W_h) and $f_h \in W_h'$ is a discrete right hand side. Of course, the discrete function space V_h and W_h should be chosen such that $V_h \to V$ and $W_h \to W$ as $h \to 0$.
The abstract form (2) can reflect standard finite element discretizations, if V_h, W_h are globally continuous, piecewise polynomial subspaces of V, W. But also Petrov Galerkin discretizations, discontinuous Galerkin approximations, or finite volume schemes can be represented in the given form for a suitable choice of discrete operators and function spaces. Even schemes which are not based on grids, like reduced basis methods which use globally defined basis functions in some other space $\tilde{W}_h$ or analytical basis functions can be cast into the framework described above.
Evolution equations of the general form

$$\partial_t v(\cdot, t) = L(v(\cdot, t))$$

with differential space operator $L : V \to W'$ can be treated by combining the framework for the stationary case with a solver for ordinary differential equations using, for example, the method of lines approach.

From the above observations we conclude that an abstract definition of discretization schemes can be obtained through a proper definiton of discrete function spaces and discrete operators. On the other hand, when it comes to the solution of the discretized problems, we would probably prefer to use vectors and matrices, rather than discrete functions and operators. Since V_h and W_h are finite-dimensional spaces, there exist isomorphisms $I_{V_h} : V_h \to \mathbb{R}^{\dim V_h}$ and $I_{W_h} : W_h \to \mathbb{R}^{\dim W_h}$ and the discrete operator L_h can be interpreted as a mapping $\tilde{L}_h : \mathbb{R}^{\dim V_h} \times \mathbb{R}^{\dim W_h} \to \mathbb{R}$ through

$$\tilde{L}_h(\tilde{v}, \tilde{w}) := \langle L_h(I_{V_h}^{-1}\tilde{v}), I_{W_h}^{-1}\tilde{w}\rangle.$$

Similarly, the right hand side $f_h \in W_h'$ can be interpreted as a mapping $\tilde{f}_h : \mathbb{R}^{\dim W_h} \to \mathbb{R}$ through

$$\tilde{f}_h(\tilde{w}) := \langle f_h, I_{W_h}^{-1}\tilde{w}\rangle.$$

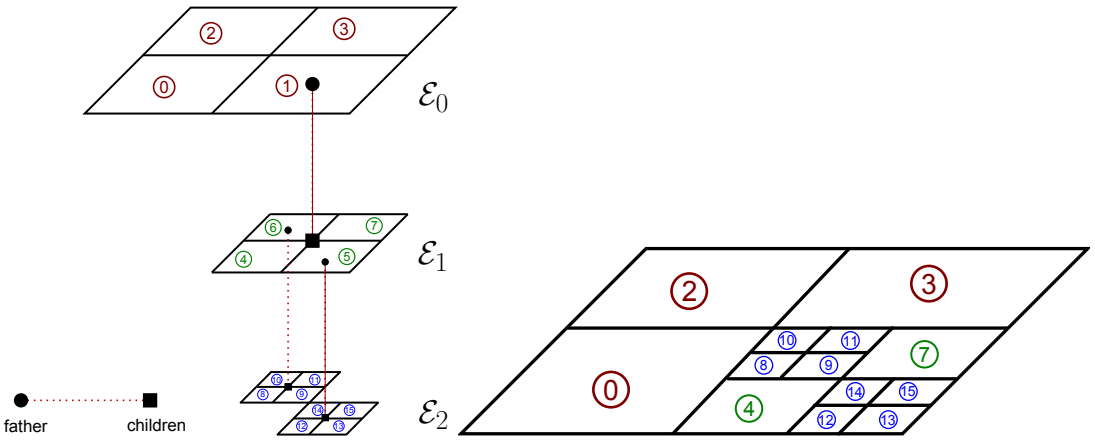
Thus, the discrete problem is equivalent to solving

$$\tilde{L}_h(\tilde{v}_h, \tilde{w}_h) = \tilde{f}_h(\tilde{w}_h) \quad \forall \tilde{w}_h \in \mathbb{R}^{\dim W_h}. \tag{3}$$

This gives us a view which can be handled with numerical linear algebra and many packages use the concept of vectors and matricies for the implementation of their numerical schemes. Using the isomorphisms I_{V_h} and I_{W_h}, we will be able to switch between both points of view. We will reflect both interpretations in our abstraction for numerical schemes in Section 2. However, as the derivation of modern numerical schemes is mostly formulated using discrete function spaces and functions, this point of view is to be the basis of our construction of the interface used in DUNE-FEM.

2 Abstract description of discretization schemes for partial differential equations

The base of our abstraction for discretization schemes is the DUNE grid interface. The theoretical description can be found in [6]. In the next subsection we will recall the essentials of the definition of a grid from this article.



(a) hierarchical grid (b) leaf grid

Figure 1: Hierarchical grid $\mathcal{H} := \{\mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2\}$. For simplicity, entities of higher codimension are not plotted. The leaf grid (according to [6, Definition 16]) is also shown. Entities from the set $\mathcal{E}_0$ (level 0) are colored in red, from $\mathcal{E}_1$ (level 1) in green, and from $\mathcal{E}_2$ (level 2) in blue.

2.1 Hierarchic grid structure

The grid interface of DUNE is described in [6]. In this section we recall the central definitions with some simplifications in comparison to the very detailed descriptions in [6]. The aim is to give a general definition of a grid which in most publications is referred to as a non-degenerate triangulation $\mathcal{T}_h$, where h denotes a characteristic grid width.
In the following we consider $\Omega \subset \mathbb{R}^w$, $w > 0$, to be the computational domain. Usually, $w = 1, 2, 3$, but higher dimensions are also allowed. We denote by $\mathcal{G}$ a discrete approximation of Ω and all objects depending on a grid will be equipped with a subscript $\mathcal{G}$.

Definition 1 (Reference Elements) *As in [6, Def. 4], we define $\mathcal{R}$ to be a finite set of reference elements.*

Remark 2 (Reference Elements implemented in DUNE) *Currently, DUNE provides reference elements for points, lines, triangles, quadrilaterals, tetrahedra, pyramids, prisms and cubes. Simplices and hypercubes are defined for arbitrary dimension.*

In the following we consider hierarchical grids in the sense of [6, Def. 13]. A hierarchical grid consists of entities distinguished by their codimension, .e.g., 0 for the elements, 1 for the edges, and 2 for the verticies of a 2d grid (cf. [6]). In Figure 1a such a 2d hierarchical grid $\mathcal{H}$ is shown, containing 3 levels created by refinement of element 1, and then by refinement of element 5 and 6. The leaf grid of $\mathcal{H}$ in the sense of [6, Definition 16] is shown in Figure 1b.

Remark 3 (Entities and geometric realization) *In contrast to [6], we do not distinguish between entity and its geometric realization, i.e., we will also consider an entity E to be a subset of $\mathbb{R}^w$, where appropriate. For a more detailed description we refer to [6, Def. 10].*

Definition 4 (Grid) *Let $\mathcal{H} := \{\mathcal{E}_0, \dots, \mathcal{E}_m\}$, $m \geq 0$, be a **hierarchical grid** in the sense of [6, Def. 13]. For a given subset $\mathcal{G} \subset \bigcup_{i=0}^{m} \mathcal{E}_i$ we define the domain $\Omega_\mathcal{G} := \bigcup_{E \in \mathcal{G}} E$. The dimension of $\mathcal{G}$ will be denoted by $d := \dim \mathcal{G} := \dim \Omega_\mathcal{G}$. As mentioned above, the elements of $\mathcal{G}$ are called entities and we denote by $\mathcal{G}^c := \{E \in \mathcal{G} \mid \operatorname{codim} E = c\}$ the entities of codimension c in $\mathcal{G}$. $\mathcal{G}$ forms a **grid** on $\Omega_\mathcal{G}$, if the following holds:*

$$E, E' \in \mathcal{G}^0, E \neq E' \implies int(E) \cap int(E') = \emptyset.$$

Definition 5 (Level of an entity) *Recalling [6, Def. 14], for a hierarchical grid $\mathcal{H} := \{\mathcal{E}_0, \dots, \mathcal{E}_m\}$ we define the* ***level function*** *$l : \bigcup_i \mathcal{E}_i \to \mathbb{N}_0$ through*

$$l(E) = i \iff E \in \mathcal{E}_i. \tag{4}$$

The ***maximal level*** *of the hierarchical grid is denoted with m.*

Remark 6 (Example Grids) *Given a hierarchical grid $\mathcal{H}$, subsets in the sense of Definition 4 are for example the level-0 grid of $\mathcal{H}$ ([6, Def. 13]) or the leaf grid of $\mathcal{H}$ ([6, Def. 16]).*

Remark 7 (Vertices and elements) *The entities $\mathcal{G}^0$ of codimension 0 are also referred to as* ***elements****; the entities $\mathcal{G}^d$ are also called* ***vertices****. Given an entity $E \in \mathcal{G}$, we denote by $\mathcal{V}_E := \{v \in \mathcal{G}^d \mid v \subset E\}$ the* ***set of vertices*** *of E.*

Definition 8 (Reference Mapping) *Given a grid $\mathcal{G}$, we assume that for each entity $E \in \mathcal{G}$ there exists exactly one reference element $\hat{E} \in \mathcal{R}$ and a diffeomorphism $F_E : \hat{E} \to E$. We call F_E the* ***reference mapping*** *of E. We denote by $\mathcal{G}^{\hat{E}}$ the set of entities with common reference element $\hat{E}$.*

Definition 9 (Father relation) *For a hierarchical grid $\mathcal{H}$ and an element $E \in \mathcal{H}$ (i.e., an entity of codimension 0), we denote by $\mathcal{C}_E$ the set of all children of E (see [6, Def. 11]). For $e \in \mathcal{C}_E$ we call E the* ***father*** *of e. We call $\mathcal{C}_E^1 := \{e \mid e \in \mathcal{C}_E \text{ and } l(e) = l(E) + 1\}$ the set of all* ***direct children*** *of E. Furthermore, we call E a* ***leaf entity*** *if $\mathcal{C}_E = \emptyset$.*

To implement numerical schemes, degrees of freedom (DoF) have to be stored and these have to be associated with the entities in a grid. This could, for example, be done as in the Finite Element toolbox ALBERTA [33], where a DoF mapping indexing arrays storing the actual DoFs is stored on the elements. In ALUGrid [36, 34, 18], there is no such thing as a DoF mapper; the numerical data itself is stored on the element structure. In a similar way DoFs are bound to elements in UG [4]. However, this leads to an unnecessary composition of grid structures with user data which dramatically decreases the flexibility of the code.
In Dune, a more general way to attach data to the grid is used: index sets associate with each entity a natural number that is unique within the set $\mathcal{G}^{\hat{E}}$. The grid interface provides two consecutive index sets, i.e., the *level index map (set)* and the *leaf index map (set)* [6, Defs. 24, 25] and a persistent index map for data transfer in adaptive computations [6, Def. 26].
In the following we give a common definition of an index set and a consecutive index set which can be used to construct DoF mappings.

Definition 10 (Index map) *Let $\mathcal{G}$ be a grid after Definition 4. Given injective mappings $\lambda_{\mathcal{G}}^{\hat{E}} : \mathcal{G}^{\hat{E}} \to \mathbb{I}$ for $\hat{E} \in \mathcal{R}$, we define the* ***index map*** *$\lambda_{\mathcal{G}}(E) = \lambda_{\mathcal{G}}^{\hat{E}}(E)$.*

Definition 11 (Index set) *Given a grid $\mathcal{G}$ in the sense of Definition 4, we call $\Lambda_{\mathcal{G}} = \{\Lambda_{\mathcal{G}}^{\hat{E}} \mid \hat{E} \in \mathcal{R}\}$ an* ***index set*** *if $\Lambda_{\mathcal{G}}^{\hat{E}} \subset \mathcal{G}^{\hat{E}} \times \mathbb{N}_0$ with $|\Lambda_{\mathcal{G}}^{\hat{E}}| = |\mathcal{G}^{\hat{E}}|$ and $\forall\, (E_1, i_1), (E_2, i_2) \in \Lambda_{\mathcal{G}}^{\hat{E}} : E_1 = E_2 \iff i_1 = i_2$, i.e., every entity E has a unique index i within the set of entities with the same reference element. We call $s^{\Lambda_{\mathcal{G}}^{\hat{E}}} := 1 + \max_{(E,i) \in \Lambda_{\mathcal{G}}^{\hat{E}}} i$, the* ***size*** *of the index set $\Lambda_{\mathcal{G}}^{\hat{E}}$.*
An index set induces an index map $\lambda_{\mathcal{G}}$ via $\lambda_{\mathcal{G}}^{\hat{E}}(E) := i \iff (E, i) \in \Lambda_{\mathcal{G}}^{\hat{E}}$.

Index sets can also provide a consecutive numbering of entities:

Definition 12 (Consecutive index set) *Given a grid $\mathcal{G}$, we call an index set $\Lambda_{\mathcal{G}}$ a* ***consecutive index set*** *for $\mathcal{G}$ if $\forall\, \hat{E} \in \mathcal{R}$: $s^{\Lambda_{\mathcal{G}}^{\hat{E}}} = |\Lambda_{\mathcal{G}}^{\hat{E}}|$, i.e., all entities with reference element $\hat{E}$ in the grid $\mathcal{G}$ are numbered from 0 to $s^{\Lambda_{\mathcal{G}}^{\hat{E}}} - 1$.*

Remark 13 (Index maps from DUNE-GRID) *Given an index map satisfying Definition 10 with a range set $\mathbb{I} \subset \mathbb{N}$, we can define an index set through $\{(E, \lambda_{\mathcal{G}}(E))\}_{E\in\mathcal{G}}$. Thus, a level index map κ_j from [6, Def. 24] naturally defines a* ***level index set*** *which forms a consecutive index set in the sense of Definition 12. Likewise, we get a consecutive* ***leaf index set*** *from the leaf index map λ described in [6, Def. 25].*
On the other hand, the persistent index maps provided by DUNE-GRID *(see [6, Def. 26]) do not induce an index set, since their range can arbitrary; however, they do satisfy Definition 10.*

For numerical evaluation of integrals on the reference elements quadratures are needed. A quadrature is a set of points and corresponding weights. The quadrature points are located in a certain reference element on which the integrand is to be evaluated. The related weights sum up to the volume of the reference element.

Definition 14 (Quadrature) *Given a reference element $\hat{E} \in \mathcal{R}$ we call a finite set of points $P_{\hat{E}} := \{\hat{\lambda} \mid \hat{\lambda} \in \hat{E}\}$ a set of* ***integration points*** *and a set $Q_{\hat{E}} := \{(\hat{\lambda}, \omega) \mid \hat{\lambda} \in \hat{E}, \omega \in \mathbb{R}$ and $\sum \omega = |\hat{E}|\}$ a* ***quadrature*** *on $\hat{E}$. A quadrature $Q_{\hat{E}}$ is called a* ***quadrature of order*** **k** *if for any polynomial function $f \in \mathbb{P}_k(\hat{E})$ the following holds:*

$$\int_{\hat{E}} f(x)\,dx = \sum_{(\hat{\lambda},\omega)\in Q_{\hat{E}}} \omega f(\hat{\lambda}).$$

Remark 15 *If no confusion is possible, we denote objects which depend on a reference element $\hat{E}$ by a superscript c for the codimension, e.g., $\Lambda^c_{\mathcal{G}}$ instead of $\Lambda^{\hat{E}}_{\mathcal{G}}$.*

2.2 Discrete functions

In this section we present an abstract framework for grid based discretization schemes. The concept of a discrete function space and a discrete function in the sense of the DUNE-FEM framework is defined. These definitions use the concepts from the previous section.
For the definition of discrete function spaces and discrete functions, we restrict ourselves to discretizations based on grids as defined in Definition 4.

Definition 16 (Function space) *By $V^{\Omega,U} = \{v : \Omega \to U\}$ we denote an arbitrary* ***function space*** *with domain $\Omega \subset \mathbb{R}^w$ and range U, $\dim U = r \in \mathbb{N}$.*

Definition 17 (Grid Function) *We say that $v \in V^{\Omega_{\mathcal{G}},U}$ is a* ***grid function*** *and for $E \in \mathcal{G}$ we call $v_E : E \to U$, $v_E := v_{|E}$ the* ***local function*** *of v on E.*

Definition 18 (Discrete function space) *Let $V_{\mathcal{G}} \subset V^{\Omega_{\mathcal{G}},U}$ be a finite dimensional subspace of a function space $V^{\Omega_{\mathcal{G}},U}$. For each element $E \in \mathcal{G}$, let $\mathcal{B}_E = (\varphi^E_i)_{i\in I_E} \subset V^{E,U}$ with $I_E \subset \mathbb{N}_0$ and let $\mu_E : I_E \to V_{\mathcal{G}}$. Then, we call $\mathcal{D}_{\mathcal{G}} := \big(V_{\mathcal{G}}, (\mathcal{B}_E)_{E\in\mathcal{G}}, (\mu_E)_{E\in\mathcal{G}}\big)$ a* ***discrete function space****, if the following conditions hold:*

1. *$\mathcal{B}_{\mathcal{G}} := \bigcup_{E\in\mathcal{G}} \mu_E(I_E)$ is a basis of $V_{\mathcal{G}}$. We call $\mathcal{B}_{\mathcal{G}}$ the* ***global base function set*** *of $\mathcal{D}_{\mathcal{G}}$.*
2. *For all elements $E \in \mathcal{G}$ and all $i \in I_E$, $\mu_E(i)$ is a continuation of φ^E_i, i.e., $\mu_E(i)_{|E} = \varphi^E_i$.*
3. *For all $\psi \in \mathcal{B}_{\mathcal{G}}$ and all elements $E \in \mathcal{G}$, we have $\psi_{|E} \in \mathcal{B}_E \cup \{0\}$.*

In this case, we call $\mathcal{B}_E$ a ***local base function set*** *and μ_E a* ***local DoF mapper****.*

Definition 19 (Discrete function) *Let $\mathcal{D}_{\mathcal{G}} := \big(V_{\mathcal{G}}, (\mathcal{B}_E)_{E\in\mathcal{G}}, (\mu_E)_{E\in\mathcal{G}}\big)$ be a discrete function space. A function $u \in V_{\mathcal{G}}$ is called a* ***discrete function****. Using the base function set of $\mathcal{D}_{\mathcal{G}}$, we have the representation*

$$u = \sum_{\psi\in\mathcal{B}_{\mathcal{G}}} u_\psi\,\psi.$$

The vector $(u_\psi)_{\psi\in\mathcal{B}_{\mathcal{G}}}$ is called the ***global DoF vector*** *of u. In the following we will also write $u \in \mathcal{D}_{\mathcal{G}}$ to denote discrete functions and to fix the corresponding global DoF vector.*

Definition 20 (Local discrete function) *For a discrete function $u \in \mathcal{D}_\mathcal{G} = \big(V_\mathcal{G}, (\mathcal{B}_E)_{E\in\mathcal{G}}, (\mu_E)_{E\in\mathcal{G}}\big)$ and an element $E \in \mathcal{G}$, we define the* ***local discrete function***

$$u_E = u_{|E} = \sum_{\psi\in\mathcal{B}_\mathcal{G}} u_\psi \, \psi_{|E} = \sum_{i\in I_E} u_i^E \, \varphi_i^E \, .$$

Using the local DoF mapper we have the relation $u_i^E = u_{\mu_E(i)}$. The vector $(u_i^E)_{i\in I_E}$ is called the ***local DoF vector*** *of u on E.*

Example 21 (Fourier space) *A simple example is given by the finite dimensional space $V_\mathcal{G}$ spanned by the basis functions*

$$\mathcal{B}_\mathcal{G} = \{\tfrac{1}{2}, \sin x, \cos x, \dots, \sin Nx, \cos Nx\}$$

for some $N \in \mathbb{N}$ (or its tensor-product equivalent). Defining $\mathcal{B}_E = \mathcal{B}_\mathcal{G}$ and taking the mapper μ_E from $I_E = \{0, 1, 2, \dots, 2N\}$ to $\mathcal{B}_\mathcal{G}$ to be $\mu_E(0) = \frac{1}{2}$, $\mu_E(2k) = \cos kx$, and $\mu_E(2k+1) = \sin kx$ we arrive at a discrete function space.

Definition 22 (Shape function) *A function $\hat\varphi_{\hat E} \in V^{\hat E, U}$, $\hat E \in \mathcal{R}$, is called a* ***shape function****.*

Example 23 (Lagrange shape functions) *For $\hat E \in \mathcal{R}$ the set of Lagrange shape functions $\mathcal{B}_{\hat E}^{L_1}$ of order 1 is given by*

$$\mathcal{B}_{\hat E}^{L_1} := \{\hat\varphi_{\hat E,i} \mid \hat\varphi_{\hat E,i} \in \mathbb{Q}_1(\hat E), \hat\varphi_{\hat E}(\hat v_j) = \delta_{i,j} \ \forall\, i, j \in I_{\hat E}^{L_1}\},$$

where $\mathcal{V}_{\hat E} = \{\hat v_i\}_{i\in I_{\hat E}^{L_1}}$ is the set of all vertices of the reference element $\hat E$. Note that for the simplicial reference elements we have $\mathbb{Q}_1(\hat E) = \mathbb{P}_1(\hat E)$.

Example 24 (Orthonormal shape functions) *For $\hat E \in \mathcal{R}$ let $\mathcal{B}_{\hat E}^{O_1} \subset \mathbb{P}_1(\hat E)$ be defined by the Gram-Schmidt procedure applied to $\{1, x_1, \dots, x_{\dim \hat E}\}$ with respect to the L^2-norm on $\hat E$.*

Definition 25 (Localized discrete function spaces) *A base function set $(\mathcal{B}_E)_{E\in\mathcal{G}}$ is called* ***localized*** *if there exists a set of shape functions $(\mathcal{B}_R)_{R\in\mathcal{R}}$ and a transformation $(\Psi_E)_{E\in\mathcal{G}}$ such that $\mathcal{B}_E = \Psi_E \circ \mathcal{B}_{\hat E} \circ F_E^{-1}$. In this case we call $\hat{\mathcal{D}}_\mathcal{G} := \big(V_\mathcal{G}, (\mathcal{B}_R)_{R\in\mathcal{R}}, (\mu_E)_{E\in\mathcal{G}}, (\Psi_E)_{E\in\mathcal{G}}\big)$ a* ***localized discrete function space*** *with the corresponding discrete function space $\mathcal{D}_\mathcal{G} := \big(V_\mathcal{G}, (\mathcal{B}_E)_{E\in\mathcal{G}}, (\mu_E)_{E\in\mathcal{G}}\big)$.*

Example 26 (Lagrange discrete function space) *Given a grid $\mathcal{G}$ of dimension d and the set of vertices $\mathcal{G}^d$ (see Remark 7), the Lagrange function space $V_\mathcal{G}^{L_1}$ is given as the span of the following basis:*

$$\mathcal{B}_\mathcal{G}^{L_1} := \{\varphi_v \mid v \in \mathcal{G}^d, \varphi_v|_E \in \mathbb{Q}_1(E) \ \forall\, E \in \mathcal{G}, \ \varphi_v(y) = \delta_{v,y} \ \forall\, y \in \mathcal{G}^d\}.$$

The local base function set is defined as

$$\mathcal{B}_E^{L_1} := \{\varphi_v \in \mathcal{B}_\mathcal{G}^{L_1} \mid \varphi_{v|_E} \neq 0\}$$

Together with the local DoF mapper $\mu_E^{L_1}(i) = \varphi_{F_E(\hat v_i)}$ we call $\mathcal{D}_\mathcal{G}^{L_1} := \big(V_\mathcal{G}^{L_1}, (\mathcal{B}_E^{L_1})_{E\in\mathcal{G}}, (\mu_E^{L_1})_{E\in\mathcal{G}}\big)$ the ***Lagrange discrete function space****.*
Notice that the Lagrange discrete function space is also a localized discrete function space since the local base function set can be constructed using the shape function set from Example 23 and the reference mappings of each entity E. We write

$$\hat{\mathcal{D}}_\mathcal{G}^{L_1} := \big(V_\mathcal{G}^{L_1}, (\mathcal{B}_R^{L_1})_{R\in\mathcal{R}}, (\mu_E^{L_1})_{E\in\mathcal{G}}, (\Psi_E)_{E\in\mathcal{G}}\big).$$

Example 27 (Discontinuous Galerkin space) *Let a grid $\mathcal{G}$ of dimension d and the set of all elements $\mathcal{G}^0$ (see Remark 7) be given. Together with the orthonormal shape function set from Example 24 the discontinuous Galerkin space $V_{\mathcal{G}}^{DG_1}$ of polynomial order 1 is given as the span of the following basis:*

$$\mathcal{B}_{\mathcal{G}}^{O_1} := \{\varphi_i^E \mid \varphi_i^E = \hat{\varphi}_i^{\hat{E}} \circ F_E^{-1}, \hat{\varphi}_i^{\hat{E}} \in \mathcal{B}_{\hat{E}}^{O_1},\ 0 \leq i < |\mathcal{B}_{\hat{E}}^{O_1}|,\ E \in \mathcal{G},\ \hat{E} \in \mathcal{R}\}.$$

We call

$$\mathcal{D}_{\mathcal{G}}^{DG_1} := \left(V_{\mathcal{G}}^{DG_1}, (\mathcal{B}_E^{O_1})_{E\in\mathcal{G}}, (\mu_E^{DG_1})_{E\in\mathcal{G}}\right)$$

*the **discontinuous Galerkin space** with the local DoF mapper $\mu_E^{DG_1}(i) = \varphi_i^E$. Obviously, the discontinuous Galerkin space is also a localized discrete function space:*

$$\hat{\mathcal{D}}_{\mathcal{G}}^{DG_1} := \left(V_{\mathcal{G}}^{DG_1}, (\mathcal{B}_R^{O_1})_{R\in\mathcal{R}}, (\mu_E^{DG_1})_{E\in\mathcal{G}}, (\Psi_E)_{E\in\mathcal{G}}\right).$$

2.3 Discrete spatial operators

Based on the concept of discrete function spaces and discrete functions described above, we are now going to introduce an abstract concept for discrete operators that can be used to describe a large class of grid based discretization schemes.

Definition 28 (Operator) *Let V, W denote arbitrary vector spaces over fields $\mathbf{K}_V$, $\mathbf{K}_W$. Then an operator $\mathcal{L} : V \to W$ is a mapping from V to W.*

The discrete analogue is the following:

Definition 29 (Discrete operator) *A discrete operator $\mathcal{L}_{\mathcal{G}}$ is an operator that maps one discrete function space into another, i.e.,*

$$\mathcal{L}_{\mathcal{G}} : \mathcal{D}_{\mathcal{G}} \to \tilde{\mathcal{D}_{\mathcal{G}}}.$$

Here, we assume that $\mathcal{L}_{\mathcal{G}}$ can be decomposed into a global operator $\mathcal{L}_{pre}$, a set of local operators $\mathcal{L}_E$ acting on local discrete functions, and a global operator $\mathcal{L}_{post}$ defined as follows

$$\mathcal{L}_{\mathcal{G}} = \mathcal{L}_{post} \circ diag\{\mathcal{L}_E, E \in \mathcal{G}\} \circ \mathcal{L}_{pre}$$

where

$$\begin{aligned}
\mathcal{L}_{pre} &: \mathcal{D}_{\mathcal{G}} \to \{V^{E,U}, E \in \mathcal{G}\},\\
\mathcal{L}_E &: V^{E,U} \to \tilde{V}^{E,\tilde{U}},\ \textit{for all } E \in \mathcal{G},\\
\mathcal{L}_{post} &: \{\tilde{V}^{E,\tilde{U}}, E \in \mathcal{G}\} \to \tilde{\mathcal{D}_{\mathcal{G}}}.
\end{aligned}$$

Note that the definition of a discrete operator allows us to combine such operators locally, e.g.,

$$\mathcal{L}_{\mathcal{G}}^2 + \mathcal{L}_{\mathcal{G}}^1 = \mathcal{L}_{\text{post}} \circ \text{diag}\{\mathcal{L}_E^2 + \mathcal{L}_E^1, E \in \mathcal{G}\} \circ \mathcal{L}_{\text{pre}}$$

if $\mathcal{L}_{\text{post}}^1 = \mathcal{L}_{\text{post}}^2 = \mathcal{L}_{\text{post}}$ is linear and $\mathcal{L}_{\text{pre}}^1 = \mathcal{L}_{\text{pre}}^2 = \mathcal{L}_{\text{pre}}$. We can even write

$$\mathcal{L}_{\mathcal{G}}^2 \circ \mathcal{L}_{\mathcal{G}}^1 = \mathcal{L}_{\text{post}}^2 \circ \text{diag}\{\mathcal{L}_E^2 \circ \mathcal{L}_E^1, E \in \mathcal{G}\} \circ \mathcal{L}_{\text{pre}}^1$$

if we have $\mathcal{L}_{\text{pre}}^2 \mathcal{L}_{\text{post}}^1 = Id$. Thus, only one grid traversal is needed also for combined discrete operators.
The definition above is very general and covers a wide range of different classes of operators. In the following we will focus on some classes of operators with special features.

2.3.1 Projection operators

Definition 30 (Projection operator) *Let $\mathcal{D}_\mathcal{G} = \big(V_\mathcal{G}, (\mathcal{B}_E)_{E\in\mathcal{G}}, (\mu_E)_{E\in\mathcal{G}}\big)$ be a discrete function space according to Definition 18 and let $\mathcal{B}^*_\mathcal{G}$ be a set of linear functionals on V with $|\mathcal{B}^*_\mathcal{G}| = |\mathcal{B}_\mathcal{G}|$ and $\mathcal{B}^*_\mathcal{G}$ is a continuation of the dual basis to $\mathcal{B}_\mathcal{G}$. Thus, we have for each $\varphi \in \mathcal{B}_\mathcal{G}$ a $\varphi^* \in \mathcal{B}^*_\mathcal{G}$ with*

$$\varphi^*(\psi) = \delta_{\varphi\psi} \;\forall\, \psi \in \mathcal{B}_\mathcal{G}\,.$$

This set of functionals defines a projection $\Pi_\mathcal{G}\colon V \to V_\mathcal{G}$ through

$$\big(\Pi_\mathcal{G}(v)\big)(\boldsymbol{x}) := \sum_{\varphi\in\mathcal{B}_\mathcal{G}} \varphi^*(v)\,\varphi(\boldsymbol{x}) \quad \text{for all } v \in V.$$

This means that the expression $\varphi^*(v)$ defines the coefficient for the basis function φ, i.e., the DoF. This can also be defined elementwise by

$$\big(\Pi_E(v)\big)(\boldsymbol{x}) := \sum_{\varphi\in\mathcal{B}_E} \varphi^*(v)\,\varphi(\boldsymbol{x}) = \big(\Pi_\mathcal{G}(v)\big)\big|_E(\boldsymbol{x})\,. \tag{5}$$

Example 31 (Lagrange Interpolation) *Let $\mathcal{D}^{L_1}_\mathcal{G}$ be the first order Lagrange discrete function space from Example 26 on a grid $\mathcal{G}$. Then we choose for $\{\varphi^*(v)\}$ the Lagrange Interpolation onto the verticies, i.e., we have $\mathcal{B}^*_\mathcal{G} = \{\varphi^*_p \mid p \in \mathcal{G}^d\}$ with*

$$\varphi^*_p(v) := v(p)\,.$$

This allows us to define the elementwise interpolation of $u \in V$ through

$$\big(\Pi^L_E(u)\big)(\boldsymbol{x}) := \sum_{i=1}^{|\mathcal{B}_E|} \varphi^E_i(\boldsymbol{x})\,u(v_i) \quad \text{for } \boldsymbol{x} \in E \text{ and } v_i \in \mathcal{V}_E. \tag{6}$$

$\Pi^L_\mathcal{G}(u)$ can be obtained as the sum over all basis functions, i.e.,

$$\big(\Pi^L_\mathcal{G}(u)\big)(\boldsymbol{x}) := \sum_{\varphi_v\in\mathcal{B}_\mathcal{G}} \varphi_v(\boldsymbol{x})\,u(v_i), \quad v \in \mathcal{G}^d\,. \tag{7}$$

Example 32 (L^2-Projection) *Let $\mathcal{D}^{DG}_\mathcal{G}$ be the discontinuous Galerkin space from Example 27 on a grid $\mathcal{G}$. Then $\varphi^*_i(\varphi_j)$ is the L^2 scalar product on the element E, i.e.,*

$$\varphi^*_i(\varphi_j) := \int_E \varphi_i\varphi_j = \int_{\hat{E}} \varphi_i\varphi_j |detDF_E|\,.$$

The projection of u onto $V^{DG}_\mathcal{G}$ is then given elementwise by

$$\big(\Pi^{DG}_E(u)\big)(\boldsymbol{x}) := \sum_{i=1}^{|\mathcal{B}_E|} \varphi^E_i(\boldsymbol{x}) \int_{\hat{E}} \varphi^E_i u\,|detDF_E| \quad \text{for } \boldsymbol{x} \in E. \tag{8}$$

$u_\mathcal{G}$ is obtained by summing over all elements in $\mathcal{G}$, i.e.

$$\big(\Pi^{DG}_\mathcal{G}(u)\big)(\boldsymbol{x}) := \sum_{E\in\mathcal{G}} \big(\Pi^{DG}_E(u)\big)(\boldsymbol{x})\,. \tag{9}$$

2.3.2 Inverse operators

With the notion of discrete operators, we may write arbitrary grid based discretization schemes as

$$\mathcal{L}_\mathcal{G}(v_\mathcal{G}) = f_\mathcal{G}.$$

As we are interested in the solution $v_\mathcal{G}$ of the discretization, we need to apply solvers in order to invert the operator $\mathcal{L}_\mathcal{G}$. Formally we may write $v_\mathcal{G} = {\mathcal{L}_\mathcal{G}}^{-1}(f_\mathcal{G})$. Thus, a solver can be interpreted as an inverse operator.

Definition 33 (Inverse operator) *If $\mathcal{L}_\mathcal{G} : \mathcal{D}_\mathcal{G} \to \tilde{\mathcal{D}}_\mathcal{G}$ is a discrete operator, we define a corresponding inverse operator $\mathcal{S}$ as*

$$\mathcal{S}_{\mathcal{L}_\mathcal{G}} : \tilde{\mathcal{D}}_\mathcal{G} \to \mathcal{D}_\mathcal{G}.$$

An inverse operator is thus initialized with a discrete operator and maps from $\tilde{\mathcal{D}}_\mathcal{G}$ to $\mathcal{D}_\mathcal{G}$. Iterative or direct solvers of linear or nonlinear systems of equations can be realized in this concept of inverse operators.

2.3.3 Linear Operator

An important type of operators are linear operators between two discrete function spaces. A linear operator $\mathcal{L}_\mathcal{G} \colon V_\mathcal{G} \to W_\mathcal{G}$ can be represented as a matrix, if basis function sets $\mathcal{B}_{V_\mathcal{G}}, \mathcal{B}_{W_\mathcal{G}}$ have been chosen.

Definition 34 (Linear Discrete Operator) *Let two discrete function spaces following Definition 18, $\mathcal{D}^V_\mathcal{G} := \left(V_\mathcal{G}, (\mathcal{B}^V_E)_{E\in\mathcal{G}}, (\mu^V_E)_{E\in\mathcal{G}}\right)$ and $\mathcal{D}^W_\mathcal{G} := \left(W_\mathcal{G}, (\mathcal{B}^W_E)_{E\in\mathcal{G}}, (\mu^W_E)_{E\in\mathcal{G}}\right)$, be given. A linear operator $\mathcal{L}_\mathcal{G}$ between these two spaces is called a **discrete linear operator**. From Definition 19 it follows that for $u \in V_\mathcal{G}$ we have a representation $u = \sum_{\psi\in\mathcal{B}^V_\mathcal{G}} u_\psi\, \psi$. Since $\mathcal{L}_\mathcal{G}$ is linear we have*

$$\mathcal{L}_\mathcal{G}[u] = \sum_{\psi\in\mathcal{B}^V_\mathcal{G}} u_\psi\, \mathcal{L}_\mathcal{G}[\psi] \ .$$

Since for $\psi \in \mathcal{B}^V_\mathcal{G}$ the function $\mathcal{L}_\mathcal{G}[\psi]$ is in $W_\mathcal{G}$ we also have a representation of the form $\mathcal{L}_\mathcal{G}[\psi] = \sum_{\omega\in\mathcal{B}^W_\mathcal{G}} \alpha_{\omega,\psi}\, \omega$ and thus $\mathcal{L}_\mathcal{G}[u] = \sum_{\omega\in\mathcal{B}^W_\mathcal{G}} \sum_{\psi\in\mathcal{B}^V_\mathcal{G}} \alpha_{\omega,\psi}\, u_\psi\, \omega$ The coefficients for the image of $v = \mathcal{L}_\mathcal{G}[u]$ are given by

$$v_\omega = \sum_{\psi\in\mathcal{B}^V_\mathcal{G}} \alpha_{\omega,\psi}\, u_\psi \ .$$

The matrix application of the matrix $A = (\alpha_{\omega,\psi})$ thus describes the transformation of the DoF of u to the DoF of $v = \mathcal{L}_\mathcal{G}[u]$.

In a similar manner as in Definition 20, where we defined for discrete functions the concept of local discrete functions, we now proceed to define local linear operators:

Definition 35 (Local Discrete Linear Operator) *Let $\mathcal{L}_\mathcal{G}$ be a discrete linear operator between two discrete function spaces $\mathcal{D}^V_\mathcal{G}$ and $\mathcal{D}^W_\mathcal{G}$ (see Definition 34). For two elements $E_r, E_c \in \mathcal{G}$ we define the **local discrete linear operator** $\mathcal{L}_{E_r,E_c}$ through the action of the operator on local discrete functions $u_{E_c} = \sum_{i\in I^V_{E_c}} u_i^{E_c}\, \psi_i^{E_c}$ of the discrete function space $\mathcal{D}^V_\mathcal{G}$. Recalling the local DoF mapper $\mu^V_{E_c}$ from the Definition 18 which satisfy $\psi_i^{E_c} = \mu^V_{E_c}(i)_{|E_c}$ a natural choice is*

$$\mathcal{L}_{E_r,E_c}[u_{E_c}] = \sum_{i\in I^V_{E_c}} u_i^{E_c}\, \mathcal{L}_\mathcal{G}[\mu^V_{E_c}(i)]_{|E_r} \ .$$

It is $v_{E_r} = \mathcal{L}_{E_r,E_c}[u_{E_c}]$ a local discrete function on the element E_r and therefore allows a representation of the form $v_{E_r} = \sum_{i\in I^W_{E_r}} v_i^{E_r}\, \omega_i^{E_r}$. Using

$$\mathcal{L}_{E_r,E_c}[\mu^V_{E_c}(j)]_{|E_r} = \sum_{i\in I^W_{E_r}} \alpha_{i,j}\, \omega_i^{E_r}$$

we obtain for $i \in I^W_{E_r}$ the following representation of the coefficients: $v_i^{E_r} = \sum_{j\in I^V_{E_c}} \alpha_{i,j}\, u_j^{E_c}$. Thus $A_{E_r,E_c} = (\alpha_{i,j})_{i\in I^W_{E_r}, j\in I^V_{E_c}}$ defines a local matrix.
Using the local DoF mappers μ^V_E and μ^W_E we compute $\alpha_{i,j} = \alpha_{\mu^W_E(i),\mu^V_E(j)}$.

2.3.4 Combined operators

As an extension of the concept of spatial operators described so far, we study the case where the spatial operator $\mathcal{L}$ has a more complex form, involving for example higher order derivatives of v or non-linearities in a non-conservative product. In this case it is possible to employ a decomposition of $\mathcal{L}$ into simpler operators. This can be done using the concept of combined operators and passes.

Definition 36 (Combined operators and passes) *Let V, W denote vector spaces and let $\mathcal{L} : V \to W$ denote an operator. We assume that we have vector spaces $V_s, W_s, s = 1, \dots, S$ satisfying $V_0 := V$, $V_s := W_s \times V_{s-1}, s = 1, \dots, S$ and $W_S := W'$, and operators*

$$L_s : V_{s-1} \to V_s, \quad s = 1, \dots, S, \quad \text{and} \quad \Pi_S : V_S \times \cdots \times V_0 \to V_S,$$

such that

$$\mathcal{L} = \Pi_S \circ L_S \circ \cdots \circ L_1.$$

*Then $\mathcal{L}$ is called a **combined operator** and L_s a **pass** of $\mathcal{L}$.*

Using the concept of discrete operators for the approximation of the operators, this concept has its discrete analogue.

Definition 37 (Discrete combined operators and passes) *Let $\mathcal{L} : V \to W$ denote a combined operator with passes $L_s, s = 1, \dots, S$. If each of the operators $L_s : V_{s-1} \to V_s$ is approximated by a discrete operator $L_{s,\mathcal{G}} : V_{s-1,\mathcal{G}} \to V_{s,\mathcal{G}}$, we may define a **discrete combined operator** $\mathcal{L}_\mathcal{G}$ via*

$$\mathcal{L}_\mathcal{G} = \Pi_S \circ L_{S,\mathcal{G}} \circ \cdots \circ L_{1,\mathcal{G}}.$$

*In this situation, $L_{s,\mathcal{G}}$ is called a **pass** of the discrete combined operator $\mathcal{L}_\mathcal{G}$.*

2.4 Time discretization of evolution equations

As an extension of the above concept for stationary problems, let us take a look at general evolution equations of the following form

$$\partial_t v(\cdot, t) = \mathcal{L}_t[v(\cdot, t)](\cdot)$$

where $\mathcal{L}_t : V \to V'$ is supposed to be a differential operator as in the stationary case. The equation above has to be understood to mean for all $\psi \in V$

$$\langle \partial_t v(\cdot, t), \psi \rangle = \langle \mathcal{L}_t[v(\cdot, t)](\cdot), \psi \rangle$$

using the dual paring on V'. Employing a discrete version $\mathcal{L}_{t,\mathcal{G}} : V_\mathcal{G} \to V_\mathcal{G}$ for the spatial operator $\mathcal{L}_t$ we arrive at a semi-discrete version of the evolution equation:

$$\partial_t v_\mathcal{G}(\cdot, t) = \mathcal{L}_{t,\mathcal{G}}[v_\mathcal{G}(\cdot, t)](\cdot)$$

This approach is termed **method of lines**. Expressing the discrete function $v_\mathcal{G}(\cdot, t)$ in terms of the basis functions in $V_\mathcal{G}$, i.e., $v_\mathcal{G}(\cdot, t) = \sum_{\varphi \in \mathcal{B}_\mathcal{G}} u_\varphi(t)\varphi(\cdot)$ we arrive at a system of ordinary differential equations for the coefficients u_φ:

$$\sum_{\varphi \in \mathcal{B}_\mathcal{G}} \frac{d}{dt} u_\varphi(t) \langle \varphi, \psi \rangle = \langle \mathcal{L}_{t,\mathcal{G}}[v_\mathcal{G}(\cdot, t)], \psi \rangle$$

for all $\psi \in V_\mathcal{G}$. This system can now be solved by employing standard numerical methods for ordinary differential equations, e.g., Runge-Kutta or multistep methods. Depending on the stability restrictions imposed by the spatial operator one can use either an explicit or an implicit method. To overcome time-step restrictions for explicit schemes while at the same time

retaining the time accuracy of explicit methods for non-restrictive terms a suitable combination of explicit/implicit solvers is sometimes the best approach. To achieve this one has to rewrite the evolution equations using two operators

$$\frac{d}{dt}v_h(t,\cdot) = \mathcal{L}_{\mathrm{expl,h}}[v_h(t,\cdot)](\cdot) + \mathcal{L}_{\mathrm{impl,h}}[v_h(t,\cdot)](\cdot) \tag{10}$$

where $\mathcal{L}_{\mathrm{impl,h}}[v_h(t,\cdot)](\cdot)$ combines all the stability restricting terms. If both operators are for example discrete combined operators, this leads to the formulation

$$\frac{d}{dt}v_h(t,\cdot) = \Pi_S[\mathcal{L}_{\mathrm{expl,S}}[\dots\mathcal{L}_{\mathrm{expl},1}[v_h(t,\cdot)]]](\cdot) + \Pi_S[\mathcal{L}_{\mathrm{impl,S}}[\dots\mathcal{L}_{\mathrm{impl},1}[v_h(t,\cdot)]]](\cdot)\ .$$

Hence, we may apply semi-implicit time discretizations — an explicit approach for $\mathcal{L}_{\mathrm{expl,h}}$ and an implicit for $\mathcal{L}_{\mathrm{impl,h}}$. The implicit part then requires the concept of discrete inverse operators to solve the arrising non-linear system.
The simplest time-discretization is given by the forward/backward Euler method. With a time-step Δt this leads to

$$\frac{v_h(t^{n+1},\cdot) - v_h(t^n,\cdot)}{\Delta t} = \Pi_S[\mathcal{L}_{\mathrm{expl,S}}[\dots\mathcal{L}_{\mathrm{expl},1}[v_h(t^n,\cdot)]]](\cdot) + \Pi_S[\mathcal{L}_{\mathrm{impl,S}}[\dots\mathcal{L}_{\mathrm{impl},1}[v_h(t^{n+1},\cdot)]]](\cdot)$$

or alternatively

$$\big(Id - \Delta t\mathcal{L}_{\mathrm{impl,h}}\big)[v_h(t^{n+1},\cdot)] = \big(Id + \Delta t\mathcal{L}_{\mathrm{expl,h}}\big)[v_h(t^n,\cdot)]$$

so that the inverse operator to $\big(Id - \Delta t\mathcal{L}_{\mathrm{impl,h}}\big)$ has to be computed. For further details and examples concerning this concept we refer to [11].

3 Abstract description of adaptivity and parallelization

So far, storage of the degrees of freedom $(u_\varphi)_{\varphi\in\mathcal{B}_\mathcal{G}}$ for a discrete function has not been specified and the DoFs can be stored in any container. In the following we will focus on the common case of a consecutive storage $(u_i)_{i\in I_\mathcal{G}}$ with $I_\mathcal{G} = \{0,\dots,|\mathcal{B}_\mathcal{G}|-1\} \subset \mathbb{N}_0$. Using an index set $\Lambda_\mathcal{G}$ (see Definition 11) a bijection $\mu_\mathcal{G}\colon \mathcal{B}_\mathcal{G} \to I_\mathcal{G}$ can be obtained.

Example 38 (Lagrange mapper) *Given a grid $\mathcal{G}$ with dimension d and the set of vertices $\mathcal{E}^d$ (see Remark 7) then the Lagrange function space $V_\mathcal{G}^{L_1}$ is given by Example 26. Using the index set $\Lambda_\mathcal{G}^d$ we construct*

$$\mu_\mathcal{G}^{L_1}(\varphi_v) = \lambda_\mathcal{G}^d(v)$$

.

Example 39 (Discontinuous Galerkin mapper) *Consider the discontinuous Galerkin space $V_\mathcal{G}^{DG_p}$ of order p (see Example 27 for $p=1$) for a grid $\mathcal{G}$ of dimension d. The vector $(u_i)_i$ has to be organized in blocks $b_{\hat{E}}$ each containing the DoFs for all entities of a given reference element $\hat{E}$. Denote the size of local base function set by m_p; note that in the case of a discontinuous Galerkin space the local Ansatz space can be $\mathbb{P}_p$ independent of the reference element $\hat{E}$, i.e., $m_p = dim(\mathbb{P}_p)$. Then we can prescibe the bijective mapping $\mu_\mathcal{G}^{DG_1}$ by*

$$\mu_\mathcal{G}^{DG_1}(\varphi_i^E) = m_p\Big(o(b_{\hat{E}}) + \lambda_\mathcal{G}^{\hat{E}}(E)\Big) + i$$

where $o(b_{\hat{E}})$ is the offset for block $b_{\hat{E}}$. Using an ordering $<$ on the reference elements of dimension d, we can define $o(b_{\hat{E}}) = \sum_{\hat{E}'<\hat{E}} s^{\Lambda_\mathcal{G}^{\hat{E}'}}$.

3.1 Adaptive function spaces

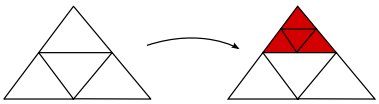
Local grid adaptivity has been proven to be a very efficient tool in decreasing numerical cost while retaining accuracy. Modifying a scheme using a fixed grid to support adaptive computations should require only minor changes. In the design of DUNE-FEM efficiency and easy use of local grid adaptivity are core concepts. To this end, each discrete function space provides a generic way for restriction and prolongation of the discrete functions. Moreover, memory allocation and DoF reordering is done transparently by a central DoF manager.
In this section, we describe the additional structures necessary to support local grid adaptation. We start off with a short overview on refinement techniques and the definition of refinement and coarsening. We then discuss the treatment of discrete functions during the adaptation phase.

3.1.1 Grid sequence

There are several well-known techniques for local grid refinement (sometimes also referred to as h-refinement). All of the techniques partition one element info several smaller ones, called *children*. These refinement techniques include *red* and *red-green* refinement [2, 8, 9, 32] as well as *bisection* refinement [3, 33, 32, 9].

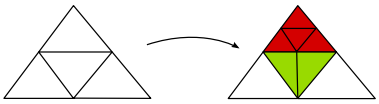
Red refinement

An element of dimension d is split into 2^d children of the same type. If the neighboring elements are not refined, *hanging nodes* will occur in the leaf grid. Each grid level will be conforming, though.



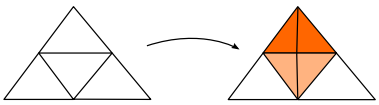
Red-green refinement

Element are refined using *red* refinement. Hanging nodes are then resolved by the socalled *green closure*. If elements of a *green closure* shall be refined themselves, they are removed and their father element is then refined *red*.



Bisection refinement

An element is bisected into exactly two children of the same type. Usually, all hanging nodes are resolved by recursively refining the neighbors. While this produces a conforming leaf grid, the grid levels need not be conforming.



A grid sequence is produced by removing child elements (coarsening) or refining elements (refinement), see also [6, Definition 23]. In the following, we assume that at a certain time there are only two coexisting grids, i.e., the grid $\mathcal{H}^n$ before the adaptation and the grid $\mathcal{H}^{n+1}$ after the adaptation.

Definition 40 (Grid sequence) *Given a hierarchical grid $\mathcal{H}^0$, an adaptive computation produces a sequence of hierarchical grids $(\mathcal{H}^n)_{n\in\mathbb{N}}$. Given a grid $\mathcal{H}^n$, the next grid in the sequence, $\mathcal{H}^{n+1}$, is created by the following two half-steps:*

- ***Coarsening:*** *The grid chooses a set $\mathcal{E}^{0,-} \subset \mathcal{H}^n$ of elements and ensures that either all or no children of any element are contained in $\mathcal{E}^{0,-}$. The elements of $\mathcal{E}^{0,-}$ are removed from $\mathcal{H}^n$ together with all subentities of $\mathcal{E}^{0,-}$ which are not subentities of $\mathcal{H}^n \setminus \mathcal{E}^{0,-}$. Let us denote this grid by $\mathcal{H}^{n+\frac{1}{2}}$.*

The set $\mathcal{E}^{0,-}$ contains elements explicitly marked for removal, if removing them will not violate the grid's closure relations. Moreover, it may contain further elements, e.g., green closure *elements that were marked for refinement.*

- ***Refinement:*** *Based on some refinement strategy, new elements are added to $\mathcal{H}^{n+\frac{1}{2}}$ along with their subentities. The result is $\mathcal{H}^{n+1}$ and we denote by $\mathcal{E}^{0,+} := \mathcal{H}^{n+1} \setminus \mathcal{H}^{n+\frac{1}{2}}$ the set of **new** entities. The new elements are chosen such that $\mathcal{H}^{n+1}$ is a valid hierarchical grid in the sense of [6, Definition 13].*

  The set $\mathcal{E}^{0,+}$ contains all children of elements that were marked for refinement. It may also contain children of elements that had to be refined due to the grid's closure relations such as the green closure or the conforming closure in bisection refinement.

*Computations on grid sequences are assumed to be structured into alternating phases of work on a fixed hierarchical grid $\mathcal{H}^n$ (**calculation phase**) and modifications of $\mathcal{H}^n$ to obtain the next grid in the sequence $\mathcal{H}^{n+1}$ (**modification phase**).*

Remark 41 *The actual order in which coarsening and refinement of the hierarchic grid are performed is not specified in the* DUNE *grid interface. In particular, the adaptation procedure described in Definition 40 as coarsening and refinement is performed by a single method called on the grid instance, turning $\mathcal{H}^n$ directly into $\mathcal{H}^{n+1}$.*

We assume in the following that a sequence $(\mathcal{H}^n)_n$ of hierarchic grids induces a sequence $(\mathcal{G}^n)_n$ where $\mathcal{G}^n \subset \mathcal{H}^n$ is a grid in the sense of Definition 4. For example, $(\mathcal{G}^n)_n$ might be the sequence of leaf grids (see [6, Definition 16]).

3.1.2 Data modification in adaptive simulations

During the modification phase of an adaptive simulation, the index sets and the corresponding DoFs have to be transferred from $\mathcal{H}^n$ to $\mathcal{H}^{n+1}$. Moreover, the discrete functions have to be restricted for entities that vanish during the modification phase. Subsequently, the data has to be prolonged to newly created entities. For these operations we will need projection operators as defined in section 2.3.

Definition 42 (Local Restriction operator) *Let $\mathcal{H}$ be a hierarchical grid and let $\mathcal{G}_1, \mathcal{G}_2 \subset \mathcal{H}$ be two grids, where for all $E \in \mathcal{G}_2 \setminus \mathcal{G}_1$ the direct children $\mathcal{C}_{E,1}$ (see Definition 9) are in $\mathcal{G}_1$. For $u_{\mathcal{G}_1} \in \mathcal{D}_{\mathcal{G}_1}$ we want to define a restriction $u_{\mathcal{G}_2} \in \mathcal{D}_{\mathcal{G}_2}$ through its operation on each $E \in \mathcal{G}_2$, i.e., we require $u_{\mathcal{G}_2|E} = u_{\mathcal{G}_1|E}$ if $E \in \mathcal{G}_1$ and*

$$u_{\mathcal{G}_2|E} = \Pi_E(\ (u_{\mathcal{G}_1|e})_{e \in \mathcal{C}_{E,1}}\ )$$

*otherwise. We then call Π_E the **local restriction operator**.*
In the more general case of the restriction during the coarsening process from $\mathcal{G}_1 \subset \mathcal{H}^n$ to $\mathcal{G}_2 \subset \mathcal{H}^{n+\frac{1}{2}}$ we might only have for all $E \in \mathcal{G}_2 \setminus \mathcal{G}_1$ $E = \bigcup_{e \in \mathcal{C}_E \cap \mathcal{G}_1} e$, i.e., not all direct children are in $\mathcal{G}_1$ but more than one level is needed. In this case Π_E is defined recursivly for all level below E to obtain the restricted data $u_{\mathcal{G}_2|E}$.

Remark 43 *We assume that the restricted function $u_{\mathcal{G}_2|E}$ is in the function space $\mathcal{D}_{\mathcal{G}_2}$, so that there are constraints on the choice of the local restriction operator Π_E, e.g., continuety must be preserved.*

Definition 44 (Local Prolongation operator) *Using the same notation as in Definition 42 we use local prolongation operators Π_E to first define the data prolongation from $E \in \mathcal{G}_1 \setminus \mathcal{G}_2$ to all $e \in \mathcal{C}_{E,1} \subset \mathcal{G}_2$; if $E = \bigcup_{e \in \mathcal{C}_E \cap \mathcal{G}_2} e$ then Π_E is again used recursively to define $u_{\mathcal{G}_2|e}$ for all $e \in \mathcal{C}_E \cap \mathcal{G}_2$.*

Remark 45 *Using the local restriction and prolongation operators data can be projected from any two grids $\mathcal{G}_1, \mathcal{G}_2 \subset \mathcal{H}$ if for all $E \in \mathcal{G}_2$ one of the following holds:*

1. $E \in \mathcal{G}_1$
2. $E = \bigcup_{e \in \mathcal{C}_E \cap \mathcal{G}_1} e$
3. *there exists a* $F \in \mathcal{G}_1$ *with* $E \in \mathcal{C}_F$

In the case of the adaptation cycle, (1) and (2) are true during coarsening and (1) and (3) hold during grid refinement. To define the projected function $u_{\mathcal{G}} \in \mathcal{G}_2$ *only local data from the direct children or the father is required.*

3.1.3 DoF storage during adaptive simulations

The consecutive index maps provided by the DUNE-GRID interface can only be used in the calculation phase and do not provide a valid index in the modification phase. To access data in any phase, the DUNE-GRID interface also provides persistent index maps assigning the same index information to an entity that is contained in both grids, $\mathcal{H}^n$ and $\mathcal{H}^{n+1}$ (see [6, Definition 26]). Since persistent indices are not necessarily natural numbers, they cannot be used to store data in a vector-like structure with constant time access or in the consecutive storage we are focusing on in this section. This means that user data that is stored in arrays in the calculation phase has to be transferred to other containers that can be used with the persistent index map. Clearly, this data transfer will increase computational cost and memory usage. However, the efficient transfer of user data from $\mathcal{H}^n$ to $\mathcal{H}^{n+1}$ is crucial for some algorithms, such as an adaptive explicit finite volume scheme, where computation time of one time step has a non-negligible part that comes from the adaptation process.
Using the functionality of the DUNE grid interface one can overcome this problem by implementing a new *persistent index set* (mapping into the natural numbers) that combines the index set features with the persistence attribute. The persistent index set is perfectly suitable for a time-explicit scheme like an explicit finite volume scheme. For implicit schemes however, non-consecutive index sets are not a good option since holes in the data arrays can dramatically increase computational costs, for example when matrix-vector operations come into play.
If the computational cost in the computation phase and the modification phase are comparable a consecutive and also persistent index set is desirable. Now the following problem arises: the presistency condition that an index stays the same for an entity that is contained in $\mathcal{H}^n$ and $\mathcal{H}^{n+1}$ cannot be satisfied if the index set is to be consecutive. Since it might be necessary to reuse an index i of an entity E which has been removed to ensure consecutivity, a persistent entity has to be assigned the index i which would break persistency.
Therefore, the persistence attribute in DUNE-FEM is defined slightly different from the persistence attribute in [6, Definition 26]. In DUNE-FEM, persistence means that an index set is capable of handling changes caused by grid modifications. To ensure this, we only need that if an index is changed the information of how it was changed is still available in the new grid.

Definition 46 (Persistent index set) *Given a hierarchic grid sequence* $(\mathcal{H}^n)_n$*, the sequence of index sets* $(\Lambda^n_{\mathcal{G}})_n$ *corresponding to the sequence* $(\mathcal{G}^n)_n$ *is called* ***persistent*** *if for any* n

1. $\Lambda^n_{\mathcal{G}}$ *is an* ***index set*** *after Definition 11, and*
2. *There exists a function* $\xi^n_{\mathcal{G}} : \mathbb{N}_0 \to \mathbb{N}_0$ *such that* $\xi^n_{\mathcal{G}}(\lambda^{n+1}_{\mathcal{G}}(E^c)) = \lambda^n_{\mathcal{G}}(E^c)$ *for all* $E^c \in \mathcal{G}^n \cap \mathcal{G}^{n+1}$*, i.e., if an entity is contained in both grids then its index in* $\mathcal{G}^n$ *can be traced in* $\mathcal{G}^{n+1}$*.*

Remark 47 *In the case that the index map induced by an index set is persistent in the sense of* DUNE-GRID*,* $\xi^n_{\mathcal{G}}$ *is the identity.*

Definition 48 (Consecutive persistent index set) *A persistent sequence of index sets* $(\Lambda^n_{\mathcal{G}})_n$ *is called a sequence of* ***consecutive persistent index sets*** *if each index set* $\Lambda^n_{\mathcal{G}}$ *is consecutive in the sense of Definition 12 during the calculation phase.*

Remark 49 *A sequence of consecutive persistent index sets provide valid indicies during the modification phase but need not be consecutive in this stage. Notice that for a sequence of consecutive persistent index sets the mapping $\xi_{\mathcal{G}}^n$ from Definition 46 is more complicated than for index sets that are only persistent.*

In the following we will use the term (consecutive) persistent index set to denote a sequence of (consecutive) persistent index sets. Consequently, we use $\Lambda_{\mathcal{G}}$ instead of $\Lambda_{\mathcal{G}}^n$.

Remark 50 (Consecutive persistent index sets in the modification phase) *As stated in Definition 48 the consecutive attribute is only needed in the calculation phase. This is reasonable since no matrix-vector like operations are needed during the modification phase. Therefore, consecutive index sets only provide this attribute in the calculation phase where it is actually needed.*

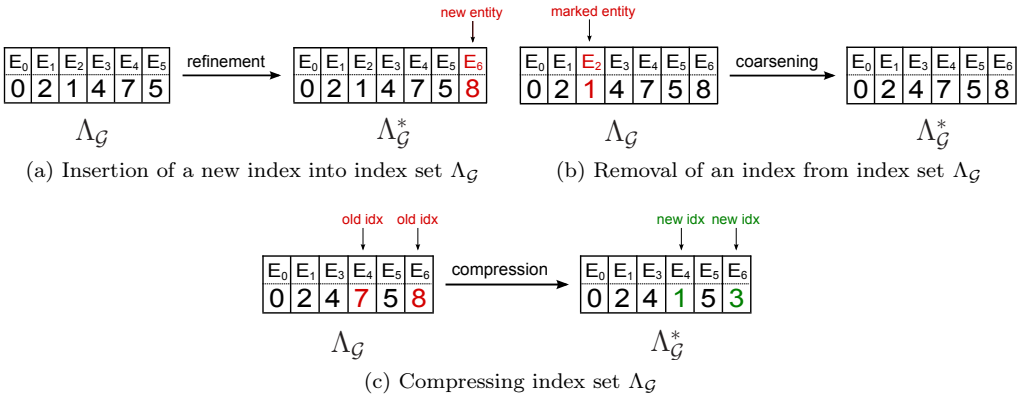
Using a persistent index set, data can be stored in containers with constant time access in both the calculation and the modification phase and no data transfer between different containers is required.
To support data restriction and prolongation, additional indices have to be inserted into persistent index sets. On data restriction one needs to create new indices and data storage for the fathers of elements that might vanish during the following adaptation step. The same applies to elements that have been newly created during the previous adaptation cycle.
In the following we sketch how a sequence of (consecutive) persistent index sets can be obtained. The whole adaptation cycle consists of the steps:

1. Before the coarsening of the grid indicies are added to the index set (see Algorithm 51).
2. Data is restricted.
3. After the adaptation phase indicies for new entities are added to the index set (see Algorithm 51).
4. Data is prolonged.
5. Indicies which are not required anymore are removed (see Algorithm 53). This index set is denoted by $\Lambda_{\mathcal{G}}^*$.
6. If a consecutive index set is required, $\xi_{\mathcal{G}}$ is constructed and the index set $\Lambda_{\mathcal{G}}^*$ is compressed to obtain the new index set (see Algorithm 55). In the case of a non-consecutive index set, $\xi_{\mathcal{G}}$ is the identity and $\Lambda_{\mathcal{G}}^*$ can be used as new index set.

Given an index set $\Lambda_{\mathcal{G}}$ the following algorithms of how to insert and remove an index or of how to compress an index set works for any $\Lambda_{\mathcal{G}}^{\hat{E}}$ in $\Lambda_{\mathcal{G}}$ and any reference element $\hat{E}$. Thus, we use the abbreviation $\Lambda_{\mathcal{G}}$ for $\Lambda_{\mathcal{G}}^{\hat{E}}$.

Algorithm 51 (Inserting indices into an index set) *Assume that a grid $\mathcal{G}$ and an index set $\Lambda_{\mathcal{G}}$ are given. If a new entity E^* is inserted, we have to define a modified index set $\Lambda_{\mathcal{G}}^* := \Lambda_{\mathcal{G}} \cup \{(E^*, s^{\Lambda_{\mathcal{G}}})\}$ for the new grid $\mathcal{G}^* := \mathcal{G} \cup \{E^*\}$. The index for the new entity is taken to be the current size of the index set $\Lambda_{\mathcal{G}}$. The new size of the index set $\Lambda_{\mathcal{G}}^*$ is then $s^{\Lambda_{\mathcal{G}}^*} := s^{\Lambda_{\mathcal{G}}} + 1$. Note that previously consecutive index sets remain consecutive.*

Example 52 (Insertion into an index set) *Consider $\mathcal{G} := \{E_0, E_1, E_2, E_3, E_4, E_5\}$. A valid index set for $\mathcal{G}$ is, for example, $\Lambda_{\mathcal{G}}^0 := \{(E_0, 0), (E_1, 2), (E_2, 1), (E_3, 4), (E_4, 7), (E_5, 5)\}$. According to Definition 11 the size of this index set is $s^{\Lambda_{\mathcal{G}}} = 8$ which is the maximal index plus one. Suppose that entity E_6 has been inserted into $\mathcal{G}$ turning it into $\mathcal{G}^*$. Following Definition 51 inserting a new index will produce $\Lambda_{\mathcal{G}}^* := \{(E_0, 0), (E_1, 2), (E_2, 1), (E_3, 4), (E_4, 7), (E_5, 5), (E_6, 8)\}$ since we insert the previous size as a new index. The size of the new index set is $s^{\Lambda_{\mathcal{G}}^*} = 9$. See also Figure 2a for an illustration.*



(a) Insertion of a new index into index set $\Lambda_{\mathcal{G}}$

(b) Removal of an index from index set $\Lambda_{\mathcal{G}}$

(c) Compressing index set $\Lambda_{\mathcal{G}}$

Figure 2: Modification of an example index set.

Algorithm 53 (Removing indices from an index set) *Given a grid $\mathcal{G}$ and an index set $\Lambda_{\mathcal{G}}$. If an entity is removed from the grid, i.e., $\mathcal{G}^* := \mathcal{G} \setminus \{E^*\}$, $E^* \in \mathcal{G}$, then the index i^* for E^* has to be removed from the index set too. One obtains $\Lambda^*_{\mathcal{G}} := \Lambda_{\mathcal{G}} \setminus \{(E^*, i^*)\}$. The size however stays the same, except in the case that $i^* = s^{\Lambda_{\mathcal{G}}} - 1$. In general this turns a previous consecutive index set into a non-consecutive index set. Thus, data compression is required at the end of the modification phase to make the index set consecutive.*

Example 54 (Removal from an index set) *Given $\mathcal{G} := \{E_0, E_1, E_2, E_3, E_4, E_5, E_6\}$ and an index set $\Lambda_{\mathcal{G}} := \{(E_0, 0), (E_1, 2), (E_2, 1), (E_3, 4), (E_4, 7), (E_5, 5), (E_6, 8)\}$. According to Definition 11 the size of this index set is $s^{\Lambda_{\mathcal{G}}} = 9$. Suppose that E_2 will be removed due to coarsening, i.e. $\mathcal{G}^* := \mathcal{G} \setminus \{E_2\}$, then the index for E_2 is also removed from the index set to obtain $\Lambda^*_{\mathcal{G}} := \{(E_0, 0), (E_1, 2), (E_3, 4), (E_4, 7), (E_5, 5), (E_6, 8)\}$. The size of $\Lambda^*_{\mathcal{G}}$ is the same as for $\Lambda_{\mathcal{G}}$, i.e. $s^{\Lambda^*_{\mathcal{G}}} = 9$. The index set is not consecutive any longer. In Figure 2a the removal of an index is shown.*

Algorithm 55 (Compression of index sets) *Let the set $\Lambda^*_{\mathcal{G}}$ obtained by inserting and removing indicies from a consecutive index set, a new consecutive index set can be obtained by defining the mapping $\xi_{\mathcal{G}}$ and chosing $\Lambda_{\mathcal{G}} = \{(E, i) \mid (E, \xi_{\mathcal{G}}(i)) \in \Lambda^*_{\mathcal{G}}\}$. Note that the final consecutive index set the condition $s^{\Lambda_{\mathcal{G}}} = |\Lambda^*_{\mathcal{G}}|$ must be fulfilled. All indices contained in the set*

$$\Xi^{old}_{\mathcal{G}} := \{i \mid (E, i) \in \Lambda^*_{\mathcal{G}} \text{ and } i \geq |\Lambda^*_{\mathcal{G}}|\}$$

are hence going to be replaced by indices contained in the following set

$$\Xi^{new}_{\mathcal{G}} := \{j \mid (E, j) \notin \Lambda^*_{\mathcal{G}} \ \forall E \in \mathcal{G} \text{ and } j < |\Lambda^*_{\mathcal{G}}|\}.$$

*Clearly, $|\Xi^{new}_{\mathcal{G}}| = |\Xi^{old}_{\mathcal{G}}| =: s_h$ which is called the **number of holes**. The mapping $\xi_{\mathcal{G}}$ can be defined based on the natural ordering on the sets $\Xi^{old}_{\mathcal{G}}$ and $\Xi^{new}_{\mathcal{G}}$. Using the uniquely defined bijective and increasing functions $\xi^{old}_{\mathcal{G}} \colon \{0, \dots, s_h - 1\} \to \Xi^{old}_{\mathcal{G}}$ and $\xi^{new}_{\mathcal{G}} \colon \{0, \dots, s_h - 1\} \to \Xi^{new}_{\mathcal{G}}$ we define*

$$\xi_{\mathcal{G}}(\xi^{new}_{\mathcal{G}}(k)) := \xi^{old}_{\mathcal{G}}(k) \text{ for } k = \{0, \dots, s_h - 1\}.$$

*We call $\xi^{old}_{\mathcal{G}}$ the **old index mapping** and $\xi^{new}_{\mathcal{G}}$ the **new index mapping**.*

Example 56 (Compression of index sets) *The index set $\Lambda^*_{\mathcal{G}}$ from Example 54 can be turned into a consecutive index as follows. Study $\Lambda_{\mathcal{G}} := \{(E_0, 0), (E_1, 2), (E_3, 4), (E_4, 7), (E_5, 5), (E_6, 8)\}$ with size $s^{\Lambda_{\mathcal{G}}} = 9$. Following Definition 55 the set of indices that are larger or equal to $|\Lambda_{\mathcal{G}}| = 6$ is $\Xi^{old}_{\mathcal{G}} := \{7, 8\}$. These indices have to be replaced by indices $0 \leq i < |\Lambda_{\mathcal{G}}| = 6$ and that are not already used. The only possible choices are 1 and 3, i.e. $\Xi^{new}_{\mathcal{G}} := \{1, 3\}$. As illustrated in Figure 2c the consecutive index set is $\Lambda^*_{\mathcal{G}} := \{(E_0, 0), (E_1, 2), (E_3, 4), (E_4, 1), (E_5, 5), (E_6, 3)\}$. The size of $\Lambda^*_{\mathcal{G}}$ is now $s^{\Lambda^*_{\mathcal{G}}} = 6$ and $\Lambda^*_{\mathcal{G}}$ is therefore a consecutive index set according to Definition 12.*

Definition 57 (Compression of user data) *Using the mappings $\xi_{\mathcal{G}}^{old}, \xi_{\mathcal{G}}^{new}$ from Algorithm 55, we can define an **old DoF mapper** $\mu_{\mathcal{G}}^{old}$, a **new DoF mapper** $\mu_{\mathcal{G}}^{new}$, and the number of holes n_h in the DoF vector. The data compression is done by assigning all new DoFs with values from the corresponding old DoFs, i.e.,*

$$\forall\, k = 0, \dots, n_h - 1: \quad u_{\mu_{\mathcal{G}}^{new}(k)} := u_{\mu_{\mathcal{G}}^{old}(k)}.$$

Let us consider an example of how the compression of user data works.

Example 58 (Old to new DoF mapper for DG spaces) *Consider the mapper given in Example 39. The number of holes for the block $b_{\hat{E}}$ in the data vector is given by $n_h^{\hat{E}} = m_p s_h^{\hat{E}}$. Defining k, l through $i = m_p k + l$ with $0 \le i < n_h^{\hat{E}}$ and $0 \le l < m_p$, we obtain*

$$\mu_{\mathcal{G}}^{old}(b_{\hat{E}}, i) := m_p\, \xi_{\mathcal{G}}^{\hat{E},old}(k) + l.$$

In the same way we define $\mu_{\mathcal{G}}^{\hat{E},new}$ using $\xi_{\mathcal{G}}^{\hat{E},new}$.
Using the index set from Example 56 and assuming $p = 0$ which leads to $m_0 = 1$, we obtain $\mu_{\mathcal{G}}^{old}(i) = \xi_{\mathcal{G}}^{old}(i)$ and $\mu_{\mathcal{G}}^{new}(i) = \xi_{\mathcal{G}}^{new}(i)$ for all $i = 0, \dots, s_h - 1 = n_h - 1$. Now, for DoF compression the data stored in position 7 has to be moved to position 1 and the data from position 8 has to be moved to position 3, see Figure 3. After the DoF compression all holes are located at the end of the DoF vector. The new size of the DoF vector is 6 while the capacity stays 9. For further enlargement of the data vector this space can be used without reallocation of memory.

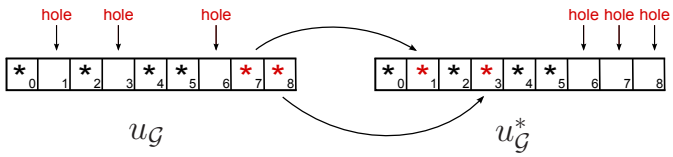


Figure 3: Compressing DoFs for $u_{\mathcal{G}}$.

Note that by using the old and new DoF mappers, time complexity for the data access is still $\mathcal{O}(1)$ since all data can be stored in arrays.

3.2 Parallelization, data exchange and dynamic load balancing

In this section we summarize the concept of distributed grid structures. As mentioned in [6, Section 5], parallel grids fulfilling the DUNE-GRID interface have to follow the 'single program multiple data' (SPMD) programming paradigm (see [14] for details) based on a suitable domain decomposition of the grid.
The domain decomposition is carried out in a two-step process. First, elements are assigned to processes (master decomposition). In a second step the decomposition for the remaining entities is determined from this master decomposition (extended decomposition). We assume that $K \ge 1$ processes are available for the parallel computation and that each process is identified by a number $p \in P := \{0, \dots, K-1\}$. Then the master decomposition is defined as follows:

Definition 59 (Master and extended decompostion) *Given a hierarchical grid $\mathcal{H}$, the master decomposition is formed by entities with codimension $c = 0$. Following [6, Definition 20], the **master decomposition** is described by the relation*

$$\mathcal{D}^0 \subseteq \mathcal{E}^0 \times P.$$

mapping the entities $\mathcal{E}^0 \subset \mathcal{H}$ to processes. If $(E, p) \in \mathcal{D}^0$ then entity E is known to process p. The set of all entities known to process p is denoted by $\mathcal{H}_p$.

For entities of higher codimension we define the ***extended decomposition*** *(see [6, Definition 22]) $\mathcal{D} \subseteq \mathcal{E} \times P$ by using the master decompostion and the subentity relation defined in [6, Definition 7]. For all $E \in \mathcal{E}^0$, $E^c \in \mathcal{E}$, $c > 0$ such that E^c is a subentity of E, the equivalence*

$$(E^c, p) \in \mathcal{D} \iff (E, p) \in \mathcal{D}^0$$

must hold for all $p \in P$, i.e., an entity E and all its subentities are always present together in a process.

The elements of a process are assigned to different classes. This is indicated by the *partition type*.

Definition 60 (Partition type, [6, Definition 21]) *The map*

$$t^0 : \mathcal{D}^0 \to \{\mathtt{i}, \mathtt{o}, \mathtt{g}\}$$

assigns a partition type $t^0(E, p)$ to entity E in process p. The three partition types are called ***interior*** (i), ***overlap*** (o), *and* ***ghost*** (g).
For entities with codimension $c > 0$, two more partition types ***border*** (b) *and* ***front*** (f) *are introduced corresponding to entities that form the boundary between interior and overlap entities and between overlap and ghost, respectively: For $0 < c \leq d$ the map*

$$t^c : \mathcal{D} \to \{\mathtt{i}, \mathtt{o}, \mathtt{g}, \mathtt{b}, \mathtt{f}\}$$

assigns a partition type $t^c(E^c, p)$ to entity E^c in process $p \in P$.

Remark 61 (Multiplicity of partition types) *Each entity E has the partition type* ***interior*** *in exactly one process, thus providing a non-overlapping decomposition of the entity set $\mathcal{E}^0$. In contrast,* ***overlap*** *elements exist in several processes because the numerical algorithm demands it explicitly (for example, overlapping Schwarz methods). Additional* ***ghost*** *elements may be necessary to ensure data consistency, for example to evaluate numerical fluxes in a finite volume or discontinuous Galerkin scheme. For more details on the partition type we refere to [6].*

The correlation between a grid in a serial computation and the same grid used in a parallel computation is given by the *master entities*. In a parallel computation all master entities form the grid that would be present in an equivalent serial computation.

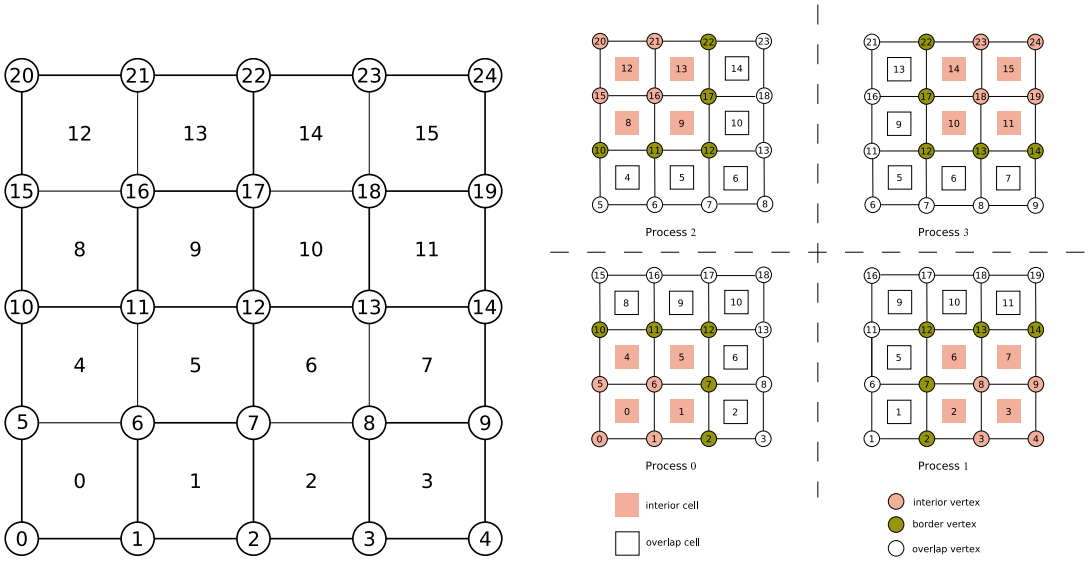
Definition 62 (Master process) *Each entity $E^c \in \mathcal{H}$ is assigned a* ***master process***

$$m(E^c) = \begin{cases} p & t^c(E^c, p) = i \\ \min\{q \mid (E^c, q) \in \mathcal{D}, t^c(E^c, q) = b\} & \text{otherwise} \end{cases}$$

We will call an entity $E^c \in \mathcal{H}|_p$ a ***master entity*** *if $m(E^c) = p$ and an entity $E^c \in \mathcal{H}|_q$ a* ***slave entity*** *if $m(E^c) \neq q$.*

Remark 63 (Distributed discrete functions) *To construct a discrete function u on a grid $\mathcal{G}$ given a distributed discrete function u_p on $\mathcal{G}_p$, the DoFs associated with an entity E are taken from $\mathcal{G}_{m(E)}$, i.e., from the master entity. Therefore, numerical algorithms have to ensure consistency in the sense that the data attached to master entities has to correspond to the data from a serial computation.*

In Figure 4a a quadrilateral grid is shown. Using this grid in a parallel computation with 4 processes a possible partitioning with one overlap layer displaying the partition types is shown in Figure 4b.



(a) serial grid (b) distributed grid with one overlap layer on 4 processors

Figure 4: Example quadrilateral grid with 16 elements and 25 vertices. For simplicity, edges are not considered.

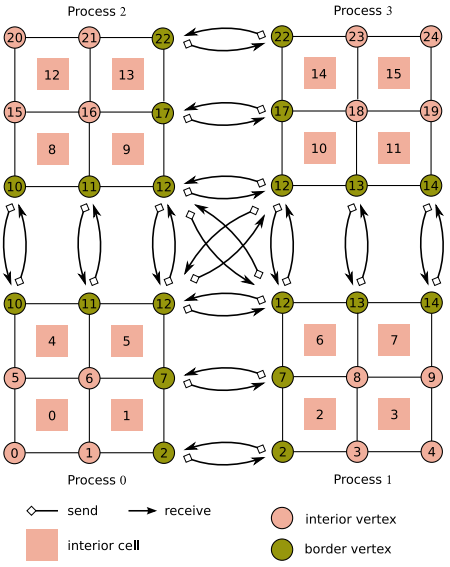
3.2.1 Communication

As described in [6, Remark 4], the data exchange is an important part of any parallel algorithm. As an example the communication is explained for different user data using the different grid partitionings presented in Figure 5.

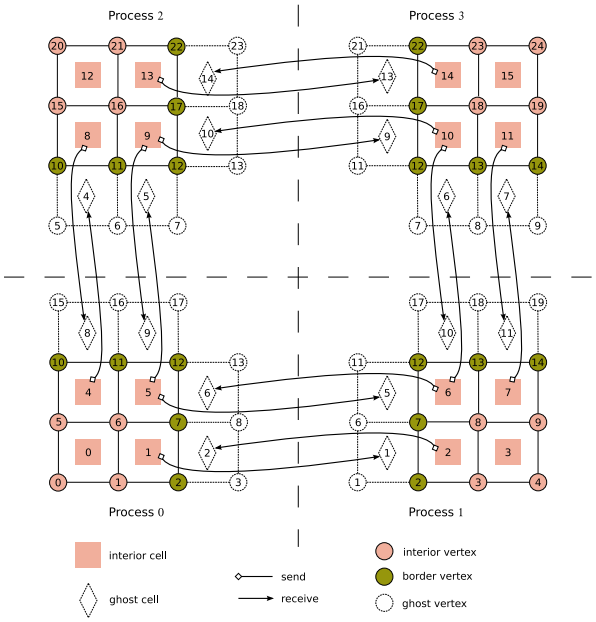
In DUNE communication means that data associated with the same entity in different processes is exchanged, and any parallel grid implementation has to support this. Formally this can be described as follows. In each process $p \in P$ select a set of source entities $\Sigma_p \subseteq \mathcal{H}|_p$ and a set of destination entities $\Delta_p \subseteq \mathcal{H}|_p$. Then a communication operation moves data associated with the entities $E^c \in \Sigma_p \cap \Delta_q$ from process p to process $q \neq p$ (forward communication) or vice versa (backward communication). In the DUNE-GRID interface predefined subsets Σ_p, Δ_p are available, see [5, Section 3.4].

Example 64 (Communication for the Lagrange space) *We study the non-overlapping decomposition shown in Figure 4a. For a discrete function $u_\mathcal{G} \in \mathcal{D}_\mathcal{G}^{L_1}$ (see Definition 26) all DoFs attached to interior entities (colored light red in Figure 5a) need not be exchanged, since these entities only exist in exactly one process. All DoFs attached to border entities (colored green in Figure 5a) have to be exchanged during a communication step, i.e., $\Sigma_p = \Delta_p = \{E \in \mathcal{G}_p \mid t^c(E,p) = b\}$. The communication path is visualized in Figure 5a with the arrows between the border entities.*

Example 65 (Communication for the Discontinuous Galerkin space) *We consider the grid $\mathcal{G}$ that is shown in Figure 4a and decomposed as described in Figure 5b, i.e., with ghost elements. For a discrete function $u_\mathcal{G} \in \mathcal{D}_\mathcal{G}^{DG}$ (see Definition 27) all DoFs attached to interior entities (colored light red in Figure 5b) that exist on other processes as ghost cells have to be exchanged. Cells only existing in exactly one process are not considered during the communication step, i.e., $\Sigma_p = \{E \in \mathcal{G}_p \mid t^0(E,p) = i\}$ and $\Delta_p = \{E \in \mathcal{G}_p \mid t^0(E,p) = g\}$. The communication path is visualized in Figure 5b by the arrows between interior and ghost elements.*



(a) Partitioning of the grid in Figure 4a without overlapping or ghost cells. Also the corresponding partition types are shown (red for interior and green for border)



(b) Decomposition of the serial grid into 4 parts including a ghost cells approach.

Figure 5: Partition types and data exchange using a border–border and interior–ghost communication

3.2.2 Load balancing

Parallel computations using local grid adaptivity can cause an unbalanced grid decomposition, e.g., much more entities are assigned to one process than to the other processes (see Figure 6). In this situation processes with little work load have to wait until the other processes have completed their computations. This delay occurs whenever synchronization of processes is required, e.g., during a communication step. An unbalanced grid partitioning thus leads to a severe reduction of parallel efficiency.
A solution to this problem is the dynamic rebalancing of work load. Once local grid adaptation leads to a situation with unbalanced partitions, a new master decomposition of the hierarchical grid is created to restore the balance of work load. Thus dynamic load balancing requires that entities with partition type `interior` in a process p can be migrated to another process q together with all attached DoFs.

Definition 66 (Repartitioning operator) *Given a hierarchical grid $\mathcal{H}$ and a set of processors P note that elements $E_0 \in \mathcal{H}$ of codimension zero are interior only on one process $p \in P$. The operator $\mathcal{B}$ describes repartitioning by assigning to each entity a new process number $q \in P$:*

$$\mathcal{B}(E) = q \in P.$$

Note that $q = p$ means that an entity is not moved at all while $q \neq p$ means the entity and all attached data have to be transferred to process q.

Remark 67 (Balance of work load) *The repartitioning operator can be defined based on graph partitioning methods (e.g., ParMETIS [27]), space filling curve methods (cf. [25]), or geometric partitioning methods.*

The definition of work load depends on different parameters. A good estimate is, for example, the number of leaf entities contained in one process (cf. [34, 18]). Alternatively, the computation time needed by one process to complete certain algorithmic steps could be used.

For a better understanding the next example describes the steps that have to be done to turn the unbalanced grid in Figure 6 into the balanced decomposition in Figure 5b.

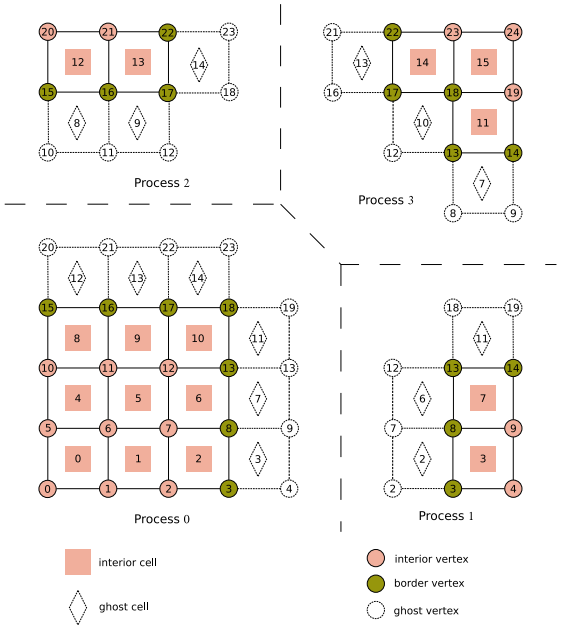


Figure 6: Unbalanced decomposition of the serial grid into 4 parts including a ghost cells approach.

Example 68 *We consider the unbalanced grid given in Figure 6. The rebalancing with the operator $\mathcal{B}$ described above could for example lead to the decomposition given in Figure 5b. To this end process* 0*, for example, has to send the topological and the geometrical information of the elements* $2, 6, 8, 9$ *and* 10 *to other processes. But also the information of neighbors has to be transferred in order to construct the necessary ghost cells. Since element* 2 *is moved to process* 1*, data attached to this element as well as data attached to all subentities also have to be copied to process* 1*.*

Remark 69 *Load balancing does not modify the serial grid $\mathcal{H}$ but only the decomposition $\mathcal{D}$. Load balancing is part of the grid modification phase, i.e., non-persistent index sets cannot be used during this process.*

4 Realization of the abstract concepts in DUNE-FEM

Following the theoretical concept from the previous sections all mathematical objects should have a representation in the implemented software. For the grid interface this has basically been described in the second paper on the DUNE grid interface [5]. Some comments and additions to this are also made in this section. The following section mostly concentrates on the implementation details of the DUNE-FEM part. An exact description of the C++ implementation of the interface classes can be found in the Appendix A. In this section we give an overview focusing on the concepts and algorithms used.

4.1 Subsets of hierarchic grids

In this section we describe classes that cast the abstract definitions from Section 2.1 into C++ objects. For classes described in the DUNE grid interface, we refer to [5]. The only exception are `IndexSets`, for which DUNE-FEM provides a richer interface. Grids, as defined in Definition 4, are not part of the DUNE grid interface. They are modeled by the class `GridPart`.

4.1.1 Index sets

Index sets form the basis for the construction of DoF mappers $\mu_{\mathcal{G}}$. They can be generated for arbitrary subsets $\mathcal{G} \subset \mathcal{H}$ of a given hierarchic grid $\mathcal{H}$ and provide an index for each entity $E \in \mathcal{G}$. Index sets in the sense of DUNE-FEM are extensions of those described by the DUNE-GRID interface and provide additional methods for enlargement and compression during adaptive computations.

Class `IndexSet` *(ref. Class 1 on page 49)*
An `IndexSet` *represents the interface for an index set* $\Lambda_{\mathcal{G}}$ *(see Definition 11).*

Class `PersistentIndexSet` $\longrightarrow$ `IndexSet` *(ref. Class 2 on page 49)*
The `PersistentIndexSet` *represents the interface for a persistent index set* $\Lambda_{\mathcal{G}}$ *(see Definition 46). This interface class automatically adds itself to the* `DofManager`*'s list of persistent index sets (see Class 30 and Class 34).*

Class `ConsecutivePersistentIndexSet` $\longrightarrow$ `PersistentIndexSet` *(ref. Class 3 on page 49)*
A `ConsecutivePersistentIndexSet` *represents the interface for a consecutive and persistent index set* $\Lambda_{\mathcal{G}}$ *(see Definition 48). In particular, this class provides methods* `numberOfHoles`*,* `oldIndex`*, and* `newIndex` *implementing the mapping* $\xi^n_{\mathcal{G}}$ *(see Definition 48).*

Example 70 (Old index – new index) *Consider the index set given in Example 56. Then the method* `numberOfHoles` *would return* 2*. The methods* `oldIndex` *and* `newIndex` *would return the following:*

`i`	`oldIndex(i)`	`newIndex(i)`
0	7	1
1	8	3

4.1.2 Grid parts

A pair $(\mathcal{G}, \Lambda_{\mathcal{G}})$ of a grid $\mathcal{G}$, in the sense of Definition 4, and an index set $\Lambda_{\mathcal{G}}$ is represented by a class implementing the `GridPart` interface.

Class `GridPart` *(ref. Class 4 on page 50)*
A `GridPart` *describes the interface for a grid* $\mathcal{G}$ *(see Definition 4) equipped with a given index set.*

The following grid parts are currently implemented in DUNE-FEM:

- The `LeafGridPart` provides a view to the leaf grid combined with the non-persistent `LeafIndexSet` (see Remark 13).
- The `LevelGridPart` is a view to a fixed level grid with the non-persistent `LevelIndexSet` (see Remark 13).

- The `AdaptiveLeafGridPart` provides a view to the leaf grid in combination with a consecutive, persistent `IndexSet` (see Definition 48). There is also a specialization of this grid part for DG methods called `DGAdaptiveLeafGridPart` providing indices only for codimension $c = 0$. This index set can be built and re-built more efficiently.
- The `FilteredGridPart` provides a view to a subset $\mathcal{G}'$ of a grid $\mathcal{G}$, described by a grid part itself. A filter is used to determine which entities are part of $\mathcal{G}'$. At the moment, the index set provided is the same as for the underlying grid part. In general, it will be non-consecutive with respect to $\mathcal{G}'$ even if it is consecutive with respect to $\mathcal{G}$.

4.2 Discrete functions

In this section we describe the DUNE-FEM interfaces that model the mathematical objects described in Section 2.2.

4.2.1 Functions and function spaces

The interface class `FunctionSpaceInterface` represents a function space $V^{\Omega_\mathcal{G},U}$ (see Definition 16). It basically defines the types for domain and range vectors and for the derivatives.

Class `FunctionSpaceInterface` *(ref. Class 5 on page 50)*
The `FunctionSpaceInterface` *represents the interface for a function space* $V^{\Omega,U}$ *(see Definition 16).*

Currently, there are two implementations of the `FunctionSpaceInterface` in DUNE-FEM:

- The `FunctionSpace`$<\mathbf{K}_D,\mathbf{K}_R,d,r>$ models functions from $\mathbf{K}_D^d \to \mathbf{K}_R^r$; all vector and matrix types are based on the `FieldVector` and `FieldMatrix` classes.
- The `MatrixSpace`$<\mathbf{K}_D,\mathbf{K}_R,d,r,c>$ models functions from $\mathbf{K}_D^d \to \mathbf{K}_R^{r\times c}$.

Grid functions are described in Definition 17. Note that they may still be analytical functions but with an elementwise representation. In DUNE-FEM, grid functions are represented by implementations of the following interface:

Class `GridFunction` $\longrightarrow$ `Function` *(ref. Class 8 on page 51)*
A `GridFunction` *represents the interface for a grid function* $v \in V^{\Omega_\mathcal{G},U}$ *(see Definition 17).*

As stated in Definition 17, a grid function has an elementwise representation, called local function. Local functions in DUNE-FEM satisfy the following interface:

Class `LocalFunction` *(ref. Class 7 on page 51)*
A `LocalFunction` *represents the interface for a local function* v_E *of a function* $v \in V^{\Omega_\mathcal{G},U}$ *on an element* E *(see Definition 17).*

4.2.2 Base function sets

A base function set represents the set $\mathcal{B}_E$ for an entity E (see Definition 18).

Class `BaseFunction` *(ref. Class 9 on page 52)*
The class `BaseFunction` *describes the interface for a single base function.*

For flexibility reasons, this interface is realized through virtual methods. The potential loss in performance is regained by caching the values and derivatives of base functions in quadrature points (see Section 4.2.3).
Base functions are always evaluated in local coordinates of the entity E. In the case of a localized base function set (see Definition 25), this is exactly what is needed to evaluate the discrete function, especially since quadratures are also mostly given in local coordinates.
The set of base functions, as described in Definition 18 is represented by the following interface:

Class `BaseFunctionSet` *(ref. Class 10 on page 52)*
A `BaseFunctionSet` *represents the interface for a base function set* $\mathcal{B}_E$ *(see Definition 18).*

4.2.3 Efficient evaluation of base functions and numerical integration

The efficient evaluation of discrete functions is essential to every numerical algorithm. Common examples include the evaluation of a function in Lagrange points or the numerical integration using a quadrature rule. In both cases, the function needs to be evaluated in an a-priori known list of points within the reference element. Especially for higher order base functions, these evaluations are expensive. This performance issue can be overcome by pre evaluating the base functions once for each such point set and storing the values (and derivatives) in a cache for later use (see Concept 72). To keep cache sizes small, this should only be done for localized base function sets (see Definition 25), since the values of their base functions depend on the reference element only.
DUNE-FEM provides two implementations of these evaluation point lists:

- The `EvaluationPointList` allows only brute force evaluation of the base functions.
- The `CachingPointList` is derived from the `EvaluationPointList` and additionally allows caching if a `CachingStorage` ist used (see Class 15).

Similarly, there are two implementations of quadratures:

- The `ElementQuadrature` is an `EvaluationPointList` that additionally provides the quadrature weights.
- The `CachingQuadrature` is a `CachingPointList` that additionally provides the quadrature weights.

The base functions discussed above might only be given as functions on a reference element $\hat{E}$. For the numerical integration over faces a special quadrature rule is needed. This quadrature is created by mapping a suitable quadrature for the reference element of the face into $\hat{E}$ (see Figure 7). Such quadratures are, for example, needed for the flux evaluation in discontinuous Galerkin methods. At the moment this concerns quadratures for the integration over elements (codimension $c = 0$) and integration over faces or intersections (codimension $c = 1$) using base functions given on the reference element. This is accomplished through the `CachingQuadrature`.

Example 71 (Quadrature on triangles) *For a quadrature exact up to order* 4 *(see [20]), the points in the reference triangle are plotted in Figure 7.*

Concept 72 (Caching of base functions) *For efficiency reasons the values of a base function in a quadrature point is cached, i.e., stored in a look-up table. This is implemented through the* `CachingStorage`*, which implements the* `BaseFunctionSet` *interface and acts as a wrapper to the real base function set. On creation of a* `CachingStorage`*, the values of all base functions are cached for each existing quadrature. Similarly, on creation of a quadrature, all* `CachingStorage`*s that have the same reference element will cache the values of all base functions in these quadrature points. Using this concept, most evaluations of base functions in local coordinates will reduce to a simple memory look-up.*
Notice that caching is enabled if and only if `CachingStorage` *is used as* `BaseFunctionSet` *and for evaluation a* `CachingPointList` *or* `CachingQuadrature` *is used.*
To evaluate local functions using base function caching, special evaluation methods are provided that take a quadrature point:

```
localFunction.evaluate(quadrature[i],val).
```

This method evaluates the local function at the i-th quadrature point using the caching mechanism. Note that calling

```
localFunction.evaluate(quadrature.point(i),val)
```

would not use caching, even if the `CachingStorage` *ist used.*

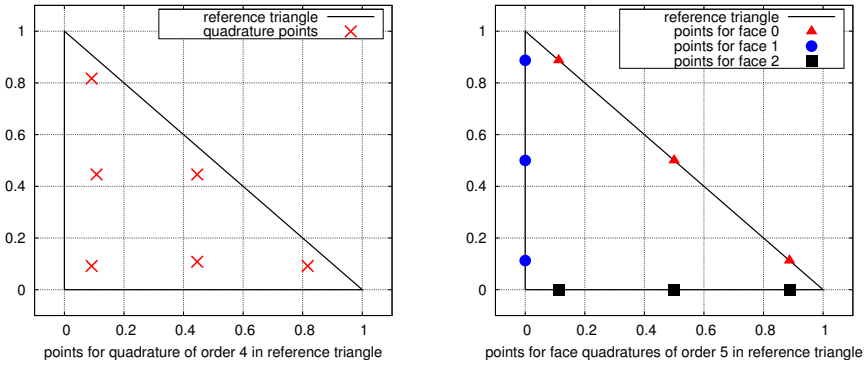


Figure 7: Points of a 4-th order `ElementQuadrature<0>` for codimension 0 on the reference triangle (left) and points of the 5-th order `ElementQaudrature<1>` for the faces 0, 1, and 2 (right).

4.2.4 Mapping degrees of freedom

A DoF mapper is needed to provide the local to global mapping for the DoFs of a discrete function. The construction of an appropriate DoF mapper is an important step in the implementation of a discrete function space. While the DoF mapper is trivial in the case of a discontinuous Galerkin space, it may become quite complicated in the case of higher order, continuous discrete function spaces, e.g., higher order Lagrange finite element spaces.

Class `DofMapper` *(ref. Class 16 on page 54)*
The `DofMapper` *represents the interface for a DoF mapper* $\mu_{\mathcal{G}}$ *(see Definition 18).*

The DoF mapper is created inside the discrete function space and can be accessed by the method `mapper` of the class `DiscreteFunctionSpace`.

4.2.5 Discrete function spaces

The interface class `DiscreteFunctionSpace` basically combines a base function set, a grid part, and a DoF mapper $\mu_{\mathcal{G}}$ forming a discrete function space.

Class `DiscreteFunctionSpace` $\longrightarrow$ `FunctionSpace` *(ref. Class 17 on page 55)*
A `DiscreteFunctionSpace` *represents the interface for a discrete function space* $\mathcal{D}_{\mathcal{G}}^{V}$ *(see Definition 18).*

The following discrete function spaces are currently implemented:

- The `LagrangeDiscreteFunctionSpace` represents the Lagrange discrete function space described in Example 26. The current implementation allows the usage of base functions of polynomial order 1 and 2 for all grid implementations and arbitrary polynomial order for twist-free grids (e.g., `SGrid`, `YaspGrid`, and `ParallelSimplexGrid`).
- The `DiscontinuousGalerkinSpace` is an implementation of the discontinuous Galerkin space described in Example 27. Orthonormal basis functions with respect to the L^2 scalar product are available up to polynomial order 8.
- The `LegendreDiscontinuousGalerkinSpace` implements a discontinuous Galerkin space using tensor product Legendre polynomials (implemented up to polynomial order 11) as base functions. The use of this space is restricted to grids containing only `cube` elements.
- The `FiniteVolumeSpace` is a simple discrete function space that can be used for first order Finite Volume methods. It provides piecewise constant Ansatz functions that evaluate to unity.

4.2.6 Discrete functions

The `DiscreteFunction` class models a function $u_\mathcal{G} \in \mathcal{D}_\mathcal{G}^V$. Since discrete function spaces have finite dimension, each discrete function is represented by a finite set of coefficients for the base functions, called the degrees of freedom (DoF) (see Definition 19).
From the programmer's point of view, a discrete function brings together the discrete function space and a certain container for the DoFs. Furthermore, a discrete function is also a grid function and therefore provides an object representing the local function u_E (see Definition 20). A local discrete function provides references to the global DoFs associated with the fixed entity E, thus allowing the direct modification of the discrete function.

Class `DiscreteFunction` $\longrightarrow$ `GridFunction` *(ref. Class 19 on page 56)*
The `DiscreteFunction` *represents the interface for a discrete function* $u_\mathcal{G} \in \mathcal{D}_\mathcal{G}$ *(see Definition 19).*

There are several implementations of discrete functions. The most useful ones are the following:

- The `AdaptiveDiscreteFunction` implements the `DiscreteFunction` interface using as DoF container the class `MutableArray<RangeFieldType>` which is a `std::vector`-like array (see [26]) implementated in DUNE-FEM. Similar to the `std::vector`, this container can be converted into a standard C array that can then be used in an external software package. An appealing feature of the `AdaptiveDiscreteFunction` is that standard C arrays can directly be turned into a discrete function, too.
- The `BlockVectorDiscreteFunction` implements the `DiscreteFunction` interface using the DUNE-ISTL `BlockVector`.
- The `VectorDiscreteFunction` provides an implementation of the `DiscreteFunction` interface using an arbitrary vector type specified by a template parameter. The vector implementation has to fulfill a `std::vector`-like interface. Derived from this class, the `ManagedVectorDiscreteFunction` can be used to store persistent data.
- The `CombinedDiscreteFunction` provides a `DiscreteFunction` combining several discrete functions of the same type to form one vector valued discrete function.

4.2.7 Local discrete functions

Since discrete functions are grid functions, they provide local functions u_E. These local discrete functions allow the modification of the global DoF associated with the entity E.

Class `LocalDiscreteFunction` $\longrightarrow$ `LocalFunction` *(ref. Class 20 on page 57)*
The `LocalDiscreteFunction` *represents the interface for a local function* u_E *of* $u_\mathcal{G}$ *(see Definition 20).*

Note that the current implementation does not distinguish between interfaces for `LocalFunction`s and `LocalDiscreteFunction`s; this will be rectified in future releases.

4.3 Discrete spatial operators

In the following we consider classes that represent the different types of operators discussed in Section 2.3. They are all derived from a common virtual base class:

Class `Operator` *(ref. Class 21 on page 58)*
The class `Operator` *prescribes the interface for a general operator* $L : V \longrightarrow W$ *that maps from one function space* V *to another function space* W.

4.3.1 Projection operators

Examples for projection operators are the *LagrangeInterpolation* described in Definition 31 or the L^2*-projection* from Definition 32. Currently, the following projection operators are implemented in DUNE-FEM:

- The operator `L2Projection` implements the L^2 projection described in Definition 32.
- The `HdivProjection` projects a discrete DG vector field into a discontinuous vector field with continuous normal components over element faces (see [7]).
- The `LagrangeProjection` projects a possibly discontinuous function into a continuous Lagrange space. For the special case of continuous data, this coincides with the Lagrange interpolation (see Definition 31).

4.3.2 Inverse operators

DUNE-FEM provides various implementations of inverse operators (seee Definition 33). The interface looks as follows:

Class `InverseLinearOperator` $\longrightarrow$ `Operator` *(ref. Class 22 on page 59)*
Given a linear `Operator` L*, the class* `InverseLinearOperator` *describes the interface for the inverse operator* $L^{-1} : \mathcal{D}_\mathcal{G} \longrightarrow \mathcal{D}_\mathcal{G}$ *(see Definition 33).*

Several inverse linear operators are currently implemented, including a `ConjugateGradientSolver` and Orthogonal Error Methods (OEM) based on the `BLAS` library.

- The `ConjugateGradientSolver` implements a CG solver using the scalar product provided by the discrete function.
- `OEMCGOp` implements an inverse operator using a BLAS based CG solver including preconditioning.
- `OEMBICGSTABOp` implements an inverse operator based on a preconditioned Bi-CGSTAB solver.
- `OEMGMRESOp` uses a preconditioned GMRES solver.

For the OEM inverse operators, SSOR preconditioning is available. To use these inverse operators, the operator L additionally has to fulfill the following interface:

Class `OEMMatrix` *(ref. Class 23 on page 59)*
Any matrix that shall be inverted using the OEM solvers has to satisfy the interface `OEMMatrix`*.*

Furthermore, DUNE-FEM provides inverse operators based on the linear solvers from DUNE-ISTL (see [10]). Bindings for the direct solver from the UMFPACK library (see [15], [41]) are also available.

- `ISTLCGOp` uses the preconditioned `CGSolver` from DUNE-ISTL.
- `ISTLBICGSTABOp` uses the preconditioned `BiCGSTABSolver` from DUNE-ISTL.
- `ISTLGMRESOp` uses the preconditioned `GMRESSolver` provided by newer versions of DUNE-ISTL.
- `UMFPACKOp` implements the interface to the UMFPACK direct solver.

For the DUNE-ISTL inverse operators the following preconditioning methods from DUNE-ISTL are available: SOR, SSOR, Gauß-Seidel, Jacobi, ILU(0), and ILU(n) (see [10]). To use the DUNE-ISTL based inverse operators, the operator L additionally has to fulfill the following interface:

Class `ISTLMatrix` *(ref. Class 24 on page 59)*
Any matrix that shall be inverted using the solvers from DUNE-ISTL *has to satisfy the interface* `ISTLMatrix`.

4.3.3 Linear operators

In DUNE-FEM, linear operators are represented by classes satisfying the following interface:

Class `LinearOperator` $\longrightarrow$ `Operator` *(ref. Class 26 on page 60)*
The class `LinearOperator` *describes the interface for linear discrete operators in* DUNE-FEM *(see Definition 34). These linear discrete operators are representable by a matrix.*

For the matrix representing the linear operator, different implementations, e.g., the `BCRSMatrix` from DUNE-ISTL, can be used. On-the-fly implementation are also possible.
A `LocalLinearOperator` provides access to all matrix entries that are associated with a certain entity; this can be seen as the extension of the `LocalDiscreteFunction` interface to linear operators.

Class `LocalLinearOperator` *(ref. Class 25 on page 59)*
The class `LocalLinearOperator` *describes the interface for local matrices (see Definition 35).*

Currently, the following linear operator implementations are provided:

- `SparseRowMatrixOperator` is based on a sparse row matrix. This operator can be used with the OEM inverse operators.
- `ISTLMatrixOperator` implements a linear operator storing a DUNE-ISTL `BCRSMatrix` which implements a blockwise *compressed row storage* concept. This operator can be used with the ISTL solvers.

4.3.4 Combined operators

For the discretization of evolution equations containing higher order derivatives the concept of combined operators from Definition 36 in Section 28 is realized by `Passes` in DUNE-FEM.

Class `Pass` $\longrightarrow$ `Operator` *(ref. Class 28 on page 61)*
The class `Pass` *provides an interface for passes of combined operators (see Definition 37).*

Passes are organized as a statically linked list of operators. The root of the pass list is the class `StartPass`. Currently, the following pass implementations are available in DUNE-FEM:

- `LocalDGPass` is an implementation of a DG discretization for first order evolution equations.
- `DGElliptPass` is an implementation for solving elliptic partial differential equations within a pass list.
- `InsertFunctionPass` is a pass to insert parameters into a pass list containing information obtained in other parts of the program, for example, in other pass lists.

Details on the implementation of the pass concept can be found in [11].

4.4 Time discretization

For solving non-stationary problems $\partial_t u = \mathcal{L}_\mathcal{G}[u]$ in DUNE-FEM, the method of lines can be used (see Section 2.4). The technical realization of a time sequence is handled by the class `TimeProvider`. It provides the simulation time and manages the time step size including synchronization in parallel simulations.

Class `TimeProvider` *(ref. Class 41 on page 66)*
The class `TimeProvider` *defines the interface for classes providing a simulation time, a time step size, CFL number and so on, for the use in non-stationary simulations.*

The interface for the discrete spatial operator $\mathcal{L}_\mathcal{G}$ is as follows:

Class `SpaceOperator` $\longrightarrow$ `Operator` *(ref. Class 42 on page 66)*
The class `SpaceOperator` *inherits the* `Operator` *class, specifying that $V = W$, i.e., $\mathcal{L}_\mathcal{G} : V \longrightarrow V$ for some discrete function space V. This operator also represents the interface for spatial discretization operators used with ODE solvers.*

For the implemented ODE solvers the interface is described by the class:

Class `OdeSolver` *(ref. Class 43 on page 67)*
The class `OdeSolver` *describes the interface for an ODE solver.*

Currently, the following ODE solver implementations are available:

- `ExplicitRungeKuttaSolver` is an implementation of the explicit Runge Kutta solvers described in [13, 24] up to order 4.
- The `ExplicitOdeSolver`[1] is a wrapper for the explicit ODE solvers from the `parDG` framework [19]. Solvers up to order 4 are available.
- The `ImplicitOdeSolver`[1] is a wrapper for the implicit ODE solvers from the `parDG` framework [19]. Solvers up to order 3 are available.
- The `SemiImplicitOdeSolver`[1] is a wrapper for the semi-implicit ODE solvers from the `parDG` framework [19]. Solvers up to order 3 are available.

Class `ODE solvers` $\longrightarrow$ `OdeSolver` *(ref. Class 44 on page 67)*
The ODE solvers `ExplicitRungeKuttaSolver`, `ExplicitOdeSolver`, *or* `ImplicitOdeSolver` *have exactly the same constructor parameter list. For the* `SemiImplicityOdeSolver` *two instead of one spatial discretization operator has to be provided.*

4.5 Data I/O, check pointing, and visualization of discrete functions

In DUNE-FEM several classes handle input-output (I/O) or visualization of discrete functions. These are

- The class `DataOutput` provides functionality for writing a `DiscreteFunction` to a file. Several formats are supported such as `XDR`, `VTK`, and `Gnuplot`.
- The class `CheckPointer` provides a check pointing functionality for writing and reading data to and from disk (using the class `DataOutput`) enabling a program to resume from a previously saved state.

  Data of objects that should be persistent are managed by the class `PersistenceManager`. All objects registered to this class are saved when a check point is written. On restart, the data is restored consistently.

[1] Due to implementation details of the `parDG` package, the field type of the discrete function space $\mathcal{D}_\mathcal{G}$ is restricted to `double` when the `ExplicitOdeSolver`, the `ImplicitOdeSolver`, or the `SemiImplicitOdeSolver` are used.

Table 1: Default communication interfaces

Discrete function space	Default communication interface
`DiscontinuousGalerkinSpace`	`InterorBorder_All_Interface`
`FiniteVolumeSpace`	`InterorBorder_All_Interface`
`LagrangeDiscreteFunctionSpace`	`InterorBorder_InteriorBorder_Interface`
`LegendreDiscontinuousGalerkinSpace`	`InterorBorder_All_Interface`

- For parameter handling throughout the code, the singleton class `Parameter` provides an easy-to-use mechanism to access program parameters from every place in the code.

DUNE-FEM provides several possibilities for data visualization.

- The implementation of the GRAPE h-Mesh interface for on-line visualizations using the GRAPE library [23] is provided. The entire grid hierarchy is retained. If numerical data is written in format `XDR` the data can also be visualized with GRAPE as a time sequence.
- The DUNE grid interface provides the class `VTKWriter` which is able to write user data and the grid $\mathcal{G}$ in a format readable by programs based on the *Visualization Tool Kit* (VTK) [43]. Examples are *VisIt* [42] or *ParaView* [39]. The DUNE-FEM class `VTKIO` extends the `VTKWriter` so that `DiscreteFunctions` can easily be stored in VTK format.
- A simple output routine also allows to write discrete functions in a format usable with `gnuplot` [37].

5 Adaptivity and parallelization in DUNE-FEM

After we have focused on the interface classes used for constructing numerical schemes, we now describe the classes realizing the concepts for parallel and adaptive computations from Section 3.

5.1 Parallelization and data exchange

The DUNE grid interface provides methods for doing data exchange in parallel computations. For all examples considered in this framework, this interfaces was sufficient for data communication in parallel runs. Every discrete function space implementation creates an internal object handling the communication. Data communication is initiated by calling the method `communicate` on the discrete function space (see Class 17).
Furthermore, each discrete function space provides a default communication interface (see Table 1). The default communication direction is always `ForwardCommunication`. A typical communication for a piecewise discontinuous space is described in Example 65 (see also [5]).
For efficiency reasons we combine the possibilities presented by the DUNE grid interface with a suitable caching mechanism. When solving a linear system using an iterative solver one has to communicate data in each iteration step of the solver. Using the communication interface of a DUNE grid this would involve iteration over grid elements in some sense, depending of course on the grid implementation. For solving linear problems efficiently the iteration steps of the linear solver have to be decoupled from any iterations over the discretization grid, if possible. Otherwise, as for example shown in Section 7.1.1, optimal performance cannot be achieved.
The DoF dependency pattern required for communications is generated using the default communication interface provided by the DUNE-GRID interface. All DoFs indicies belonging to an entity visited during this communication are stored in a look-up table. Then later communications do not involve any grid traversal, since all global DoF indicies are known and data vectors can be accessed directly.
Actual communication is handled by a communication manager which is an implementation of the following interface:

Class `CommunicationManager` *(ref. Class 29 on page 61)*
The `CommunicationManager` *is in charge of doing communications in parallel simulations. We group all communications for discrete functions belonging to the same discrete function space since the communication pattern is in most cases the same. The user still has the flexibility to exchange the data in a different way if needed. Each discrete function space defines a default communication pattern and an operation to be formed on the data.*

Currently there are two implementations of communication managers.

- `DefaultCommunicationManager` is an implementation of the `CommunicationManager` using the standard communication features provided by the DUNE grid interface.
- `CachedCommunicationManager` is an implementation of the `CommunicationManager` caching the DoF dependency patterns for faster communication.

Example 73 (CachedCommunication) *Using a cached communication manager for a linear Lagrange discrete function space the communication maps for the* `border-border` *communication described in Example 64 looks the following way:*

Process 0	$M_1 := \{2, 7, 12\}$ $M_2 := \{10, 11, 12\}$ $M_3 := \{12\}$	*Process* 2	$M_0 := \{10, 11, 12\}$ $M_1 := \{12\}$ $M_3 := \{12, 17, 22\}$
Process 1	$M_0 := \{2, 7, 12\}$ $M_2 := \{12\}$ $M_3 := \{12, 13, 14\}$	*Process* 3	$M_0 := \{12\}$ $M_1 := \{12, 13, 14\}$ $M_2 := \{12, 17, 12\}$

5.2 Adaptation and load-balancing

In adaptive simulations memory readjustment is a crucial task and it becomes even more complicated in parallel simulations due to dynamic load balancing.

5.2.1 Memory management

In this section we describe some classes dealing with memory management. For a numerics code supporting local grid adaptation, it is essential that once the grid is changed all user data, e.g., the index sets created by the user and the DoF storage, is adapted. For example, in ALBERTA all data is known to the mesh to facilitate data reorganization during adaptation (see [33]). In DUNE data storage is completely decoupled from the grid. Therefore, an alternative concept is needed. In DUNE-FEM this is realized by a manager class that triggers all necessary resize and compression processes after the grid has been changed. This manager class is:

Class `DofManager` *(ref. Class 34 on page 63)*
The `DofManager` *is an implementation for the central management of memory needed for the storage of user data, i.e., the DoFs and index sets created. To ensure that only one instance of a* `DofManager` *for a hierarchic grid $\mathcal{H}$ exists, the* `DofManager` *can only be accessed via a static method* `instance(`$\mathcal{H}$`)` *which implements the* ***singleton per unique key*** *concept. Given a hierarchic grid $\mathcal{H}$, the method* `instance` *returns a reference to the associated* `DofManager`*.*

To avoid responsibility clashes it must be ensured that this `DofManager` exists only once per grid instance. To this end, the *singleton per unique key* (see for example [1]) concept has been applied.
The `DofManager` is able to manage persistent and consecutive-persistent index sets satisfying the interface:

Class `ManagedIndexSet` *(ref. Class 30 on page 62)*
The `DofManager` *needs to know all persistent and consecutive persistent index sets depending on the grid instance* $\mathcal{H}$ *the* `DofManager` *is responsible for. Therefore, a list of* `ManagedIndexSet` *are stored. These are simple wrapper classes storing references to the real index sets. The interface methods are realized via virtual methods. A* `ManagedIndexSet` *is created by the call to the method* `addIndexSet` *of the* `DofManager`*.*

The DoF storage of a `DiscreteFunction` can be either

- **(unmanaged)**, i.e., no resize is done on grid changes, or
- **(managed)**, i.e., the size of the DoF storage is adapted whenever the grid changes and DoF compression is performed, if enabled.

The class `DofStorageInterface` and `ManagedDofStorageInterface` provides the interface for an unmanaged and a managed DoF storage, respectively.

Class `DofStorageInterface` *(ref. Class 31 on page 62)*
The `DofStorageInterface` *is the interface for an unmanged DoF storage.*

Class `ManagedDofStorageInterface` $\longrightarrow$ `DofStorageInterface` *(ref. Class 32 on page 62)*
The `ManagedDofStorageInterface` *inherits the* `DofStorageInterface` *and provides an interface for a DoF storage that should be managed by the* `DofManager`*. This means during the adaptation process the memory of these DoF storages is resized and compressed if necessary (see also Definition 57).*

On creation and deletion of a managed DoF storage the `DofManager` is notified and updates its list of DoF storages.
To trigger the adaptation process, we need a further *singleton per grid* object:

Class `AdaptationManager` $\longrightarrow$ `SerialAdaptationManager, LoadBalancer` *(ref. Class 38 on page 65)*
The `AdaptationManager` *class inherits from the* `SerialAdaptationManager` *as well as from* `LoadBalancer`*. A reimplementation of the method* `adapt` *is done.*

This class is responsible for

- calling the method `adapt` on the grid (see Definition 40),
- triggering the adaptation of DoF storages through the `DofManager`,
- performing the restriction and prolongation of user data (see Definitions 42 and 44),
- redistributing data after load-balancing in parallel computations.

In short, the `AdaptationManager` manages all parts of the *modification phase* of the simulation. To transfer the data from $\mathcal{H}^n$ to $\mathcal{H}^{n+1}$, the `AdaptationManager` must be provided with an implementation of the following interface class:

Class `RestrictionProlongation` *(ref. Class 39 on page 65)*
The `RestrictionProlongation` *represents the interface for a local restriction and prolongation operator.*

It is possible to combine `RestrictionProlongation` operators for several discrete functions into a single operator. For each discrete function space, a default implementation of the `RestrictionProlongation` operators is provided using the projections described in Section 4.3.1.

Class `RestrictProlongDefault` ⟶ `RestrictionProlongaion` *(ref. Class 40 on page 66)*
The `RestrictProlongDefault` *class implements the* `RestrictionProlongation` *interface. For each discrete function space implementation there is one such default implementation of the* `RestrictionProlongation` *interface.*

The difficulty in handling data attached to a grid in DUNE is that the adaptation process described in Definition 40 is completely done by calling the method `adapt` on the grid instance. This means that all data restriction has to be finished before the method `adapt` is called and data prolongation can only be done afterwards. A generic adaptation algorithm might look as follows:

Algorithm 74 (Generic Adaptation algorithm)

1. *Switch from* ***calculation phase*** *to* ***modification phase*** *by calling the method* `preAdapt` *on the grid instance. If true was returned start the* ***restriction*** *process:*
   (a) *Reserve memory for all persistent data.*
   (b) *Insert new indices into the persistent index sets for fathers of all elements that might vanish during the adaptation.*
   (c) *Restrict all data from elements that might vanish (see Definition 42 and Example Code 8).*
2. *Adapt the grid by calling its* `adapt` *method. If true was returned start the* ***prolongation*** *process:*
   (a) *Reserve memory for all persistent data.*
   (b) *Insert indices into the persistent index sets for all newly created elements (see Definition 51).*
   (c) *Prolong all data from fathers to their newly created children (see Example Code 9).*
   (d) *Remove indices for elements that have newly created children (see Definition 53).*
3. *If either restriction or prolongation of data was required, then the compression process is activated:*
   (a) *Compress all consecutive and persistent index sets (see Definition 55).*
   (b) *Compress all data based on consecutive and persistent index sets (see Definition 57).*
   (c) *Free unused memory.*

```
hierarchicRestriction( H )
{
  for E ∈ H with l(E) = 0 do
    recursiveRestriction( E )
}
```

```
recursiveRestriction( E )
{
  if C_{E,1} ≠ ∅ then
  {
    for e ∈ C_{E,1} do
      recursiveRestriction(e)
    if e might vanish ∀ e ∈ C_{E,1} then
      data restriction for (C_{E,1}, E)
  }
}
```

Example Code 8: Generic **restriction** algorithm from DUNE-FEM

Depending on the grid implementation different adaptation techniques are availabe as summarized in Table 2. For parallel computations DUNE provides two grid implementations that are able to deal with parallel simulations as stated in Table 2.

```
hierarchicProlongation( H )
{
  for E ∈ H with l(E) = 0 do
    recursiveProlongation(E)
}
```

```
recursiveProlongation( E )
{
  for e ∈ C_{E,1} do
    if e is new then
      data prolongation for (E, e)
  for e ∈ C_{E,1} do
    recursiveProlongation(e)
}
```

Example Code 9: Generic **prolongation** algorithm from DUNE-FEM

Table 2: Available grids and refinement techniques

Grid	element	local refinement	parallel	load balancing
`AlbertaGrid` ($d = 1, 2, 3$)	`simplex`	bisection	$\surd$	—
`ALUCubeGrid` ($d = 3$)	`cube`	red	$\surd$	dynamic
`ALUSimplexGrid` ($d = 2, 3$)	`simplex`	red	—, $\surd$	—, dynamic
`ALUConformGrid` ($d = 2$)	`simplex`	bisection	—	—
`OneDGrid`($d = 1$)	`simplex`	bisection	—	—
`UGGrid` ($d = 2, 3$)	`simplex`	red-green or red	—	—
`UGGrid` ($d = 2, 3$)	`cube`	red	—	—
`UGGrid` ($d = 2, 3$)	`hybrid`	red-green	—	—
`YaspGrid`($d = 1, 2, \ldots$)	cube	—	$\surd$	static

Remark 75 `YaspGrid` *is statically load balanced on construction. The package* UG *provides parallelized grids that can be dynamically load balanced. Nevertheless, these features are not yet implemented completely in the* DUNE *grid interface implementation* `UGGrid`.

Some DUNE grid implementations, e.g., `AlbertaGrid` and `ALUGrid`, provide an alternative adaptation procedure allowing a call back from inside adaptation algorithm. The user's restriction and prolongation operator is called as soon as an element is newly created or about to vanish. The interface method on the DUNE `Grid` class has the following simple form:

```
bool adapt( restrictProlong )
```

The choice of the adaptation algorithm depends on the discretization scheme. Both methods, the generic and the callback method have their advantages and disadvantages. For example, the callback method is faster for finite volume schemes or discontinuous Galerkin methods while it can be slower for others such as higher order Lagrange finite element methods. The choice of the adaptation algorithm therefore depends on the discretization scheme.

6 Using the DUNE-FEM module

In this Section we demonstrate the use of the classes defined in the DUNE-FEM module. Details on the concepts and the interface classes can be found in the Appendices.

6.1 L^2-Projection

As an example an L^2-projection of a given analytical function f and the projection error are computed. The discrete function u in some discrete function space $\mathcal{D}_\mathcal{G}$ is defined by the equation

$$\int_{\Omega_\mathcal{G}} u\varphi = \int_{\Omega_\mathcal{G}} f\varphi \quad \text{for all } \varphi \in \mathcal{D}_\mathcal{G}\,.$$

The first code snippet (see Listing 1) shows how discrete function spaces, functions and a linear operator are constructed. An instance of a lagrange function u on the leaf grid is created. The function maps to $\mathbb{R}^r$, i.e., is vector valued if the integer constant $r > 1$. The order of the lagrange space is p given in the template argument list. The required type for the local function u_E and the local operator M_{E_r,E_c} can be easily accessed through the types of the discrete function space and the linear operator. Other types, e.g., the iterator over the grid or the type of the local base function set $\mathcal{B}_E$ are also provided by the discrete function space.

Listing 1: Constructing discrete spaces and functions

```
// type of the grid (see Class 4), where HGridType is the DUNE-GRID type of H.
typedef AdaptiveLeafGridPart<HGridType, InteriorBorder_Partition> GridPartType;
// type of the function space (see Class 5)
typedef FunctionSpace<double, double, HGridType::dimensionworld, r> FunctionSpaceType;
// type of the discrete function space (see Class 17)
typedef LagrangeDiscreteFunctionSpace<FunctionSpaceType, GridPartType, p>
  DiscreteSpaceType;
// type of the discrete function (see Class 19)
typedef AdaptiveDiscreteFunction<DiscreteSpaceType> DiscreteFunctionType;

// type of the linear operator for the mass matrix (see Class 26)
typedef SparseRowMatrixOperator
  <DiscreteFunctionType, DiscreteFunctionType, MyMatrixTraits<DiscreteSpaceType> >
  MatrixType;

// extract required types
typedef DiscreteSpaceType::RangeType RangeType;
typedef DiscreteSpaceType::IteratorType IteratorType;
typedef DiscreteSpaceType::BaseFunctionSetType BaseFunctionSetType;

typedef DiscreteFunctionType::LocalFunctionType LocalFunctionType;
typedef MatrixType::LocalMatrixType LocalMatrixType;

// construct leaf grid G from a hierarchical grid H
GridPartType grid( hgrid );
// construct lagrange discrete function space D_G
DiscreteSpaceType space( grid );

// create the analytical function f to project
MyFunction<FunctionSpaceType> f;
// create the solution u
DiscreteFunctionType u( "solution", space );
```

In Listing 2, we perform a grid walkthrough to assemble the mass matrix M with entries $M_{ij} = \int_{\Omega_\mathcal{G}} \varphi_i \varphi_j$, and the functional b on the right hand side, given by $b_j = \int_{\Omega_\mathcal{G}} f \varphi_j$.

Listing 2: Assembling mass matrix and right hand side

```
// create the functional b and the linear operator M
DiscreteFunctionType b( "functional", space );
MatrixType M( "mass_matrix", space, space );

// initialization of M and b
M.reserve();
M.clear();
b.clear();

// create some temporary storage for the values of the local base functions B_E
std::vector<RangeType> values;

// walk over the grid G
const IteratorType end = space.end();
for( IteratorType it = space.begin(); it != end; ++it )
{
  const EntityType &entity = *it;
  const GeometryType &geometry = entity.geometry();

  // obtain local views to the functional (see Class 7)
  LocalFunctionType bLocal = b.localFunction( entity );
  // obtain local operator M_{E,E} (see Class 25)
  LocalMatrixType MLocal = M.localMatrix( entity, entity );

  // obtain the local base function set B_E
  const BaseFunctionSetType &baseFunctionSet = space.baseFunctionSet( entity );
  const unsigned int numBaseFunctions = baseFunctionSet.numBaseFunctions();
```

```
  values.resize( numBaseFunctions );

  // compute the integrals ∫_E φ_i φ_j and ∫_E f φ_i using a quadrature with base function caching (see Class 14)
  typedef CachingQuadrature<GridPartType, 0> QuadratureType;
  QuadratureType quadrature( entity, 2*space.order()+1 );
  const unsigned int nop = quadrature.nop();
  for( unsigned int qp = 0; qp < nop; ++qp )
  {
    const QuadratureType::CoordinateType &x = quadrature.point( qp );
    const double weight = quadrature.weight( qp ) * geometry.integrationElement( x );

    for( unsigned int i = 0; i < numBaseFunctions; ++i )
      baseFunctionSet.evaluate( i, quadrature[ qp ], values[ i ] );

    RangeType fValue;
    f.evaluate( geometry.global( x ), fValue );
    for( unsigned int i = 0; i < numBaseFunctions; ++i )
    {
      // add ∫_E φ_i φ_j to the operator M
      for( unsigned int j = 0; j < numBaseFunctions; ++j )
        MLocal.add( i, j, weight * (values[ i ] * values[ j ]) );
      // add ∫_E f φ_i to the functional b
      bLocal[ i ] += weight * (fValue * values[ i ]);
    }
  }
}
```

In the following Listing the mass matrix is inverted to produce $u = M^{-1}b$ using a CG based inverse linear operator.

Listing 3: L^2 projection (inversion of mass matrix)

```
// construct the inverse operator M^-1 (see Class 22)
CGInverseOp<DiscreteFunctionType, MatrixType> inverseOperator( M, 1e-10, 1e-10 );
// compute solution
u.clear();
inverseOperator( b, u );
```

In Listing 4, the L^2 difference between the analytical solution f and the discrete function u is computed. Again, a grid walkthrough is performed to calculate the error on each entity using a quadrature rule. Note the easy use of the local function to evaluate u.

Listing 4: Computing the L^2 error

```
double error = 0.0;
// iterate over the grid G
for( IteratorType it = space.begin(); it != end ; ++it )
{
  const EntityType &entity = *it;
  const GeometryType &geometry = entity.geometry();

  // get local function u_E (see Class 7)
  LocalFunctionType uLocal = u.localFunction( entity );

  // compute ∫_E |u - f|^2
  typedef CachingQuadrature<GridPartType, 0> QuadratureType;
  QuadratureType quadrature( entity, 2*space.order()+2 );
  const int nop = quadrature.nop();
  for( int qp = 0; qp < nop; ++qp )
  {
    const QuadratureType::CoordinateType &x = quadrature.point( qp );
    const double weight = quadrature.weight( qp ) * geometry.integrationElement( x );

    RangeType fValue, uValue;
    f.evaluate( geometry.global( x ), fValue );
    uLocal.evaluate( quadrature[ qp ], uValue );
    error += weight * (fValue-uValue).two_norm2();
  }
}
error = std::sqrt( error );
```

6.2 Further examples

The storage for the discrete function u itself allows for automatic resizing, prolongation, and restriction. Using the adaptation manager from DUNE-FEM and the default restriction/prolongation operators on the discrete function space, the few lines of code for adapting a grid and keeping the discrete functions consistent, are given in Listing 5. The simplest approach for data communication is also shown. Calling a single method on the discrete function space results in the communication of all data required for keeping the discrete function consistent over process boundaries.

Listing 5: Adapting and communicating a discrete function

```
// type of the default restriction and prolongation operator (see Class 40)
typedef RestrictProlongDefault<DiscreteFunctionType> RestrictProlongType;
// type of the adaptation manager (see Class 38)
typedef AdaptationManager<HGridType, RestrictProlongType> AdaptationManagerType;

// create restriction and prolongation operator for u and the adaptation manager
RestrictProlongType uRestrictProlong( u );
AdaptationManagerType adaptationManager( hgrid, uRestrictProlong );

// mark grid for refinement and coarsening using some external method mark
mark( hgrid, u );

// adapt the grid with automatic restriction and prolongation of the discrete function u
adaptManager.adapt();

// communicate u using the space's default communication (see Table 1)
space.communicate( u );
```

We conclude our short survey of the module DUNE-FEM by demonstrating in Listing 6 of how to handle the discretization of evolution equations in DUNE-FEM. Assuming that a discrete spatial operator `DiscreteOperatorType` given, the construction of an ODE solver and the simple implementation of a time stepping scheme is shown.

Listing 6: Using a time stepping scheme to solve an evolution equation

```
// initialize the time provider (see Class 41)
GridTimeProvider<HGridType> timeProvider( 0, hgrid );
// create space discretization operator derived from Class 42
DiscreteOperatorType spaceOperator( order );
// construct ODE solver (see Class 43)
typedef ExplicitRungeKuttaSolver<DiscreteFunctionType> ODESolverType;
ODESolverType odeSolver( spaceOperator, timeProvider, order+1 );

// set the initial time step estimate
odeSolver.initialize( u );
// time loop
for( timeProvider.init(); timeProvider.time() < T; timeProvider.next() )
{
  timeProvider.provideTimeStepEstimate( maxTimeStep );
  odeSolver.solve( u );
}
```

7 Proof of concept

We proceed with the presentation of several benchmark problems followed by a selection of more complex test cases.

7.1 Benchmark problems

We start off with a selection of well known benchmark problems to show the possibilities and efficiency of the DUNE-FEM software package.

7.1.1 Poisson's equation

First, we consider a benchmark problem for the Poisson equation

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \subset \mathbb{R}^d, \quad d \in \{1,2,3\} \\ u &= g \quad \text{on } \partial\Omega \end{aligned} \tag{11}$$

on the domain $\Omega = [0,1]^d$. The data is chosen to yield an exact solution

$$u(\boldsymbol{x}) = e^{-10|\boldsymbol{x}|^2}.$$

Therefore, the right hand side of (11) is given by

$$f(\boldsymbol{x}) = -\Delta u(\boldsymbol{x}) = \big(20d - 400|\boldsymbol{x}|^2\big)e^{-10|\boldsymbol{x}|^2}$$

and the boundary data is simply $g = u_{|\partial\Omega}$.

Discretization

This problem is solved using a standard conforming finite elements approach.

Implementation details

For the implementation of the finite element space the `LagrangeDiscreteFunctionSpace` (see Section 4.2.5) was used. The system matrix storage type is *compressed row*, implemented by the class `SparseRowMatrixOperator` (see Section 4.3.3). The system is solved by a standard CG method without preconditioning (see class `ConjugateGradientSolver` in Section 4.3.2). During each iteration step of the CG solver a `border-border` communication has to be performed (see Example 64). Adaptation and load balancing is handled by the class `AdaptationManager` (see Section 5.2).

Numerical results

In Figure 10 the approximate solution of equation (11) on a non-conforming adaptively refined hexahedral grid is shown. The simulation used `ALUCubeGrid` and P_1 Lagrange elements. In Figure 11a the solution of (11) on a non-conforming adaptively refined triangular grid is shown. The simulation used `ALUSimplexGrid` and P_1 Lagrange elements. The local grid adaptivity uses a residual based error estimator for this problem (see for example [33]). In Figure 11b the solution of (11) on a Cartesian grid using P_6 Lagrange elements is shown.

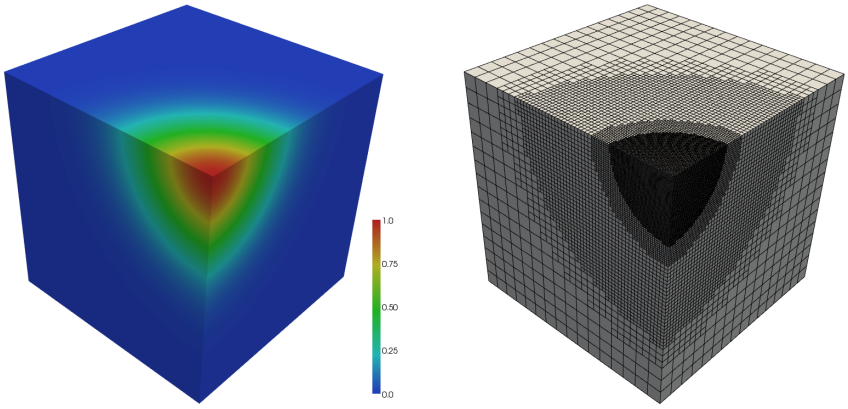
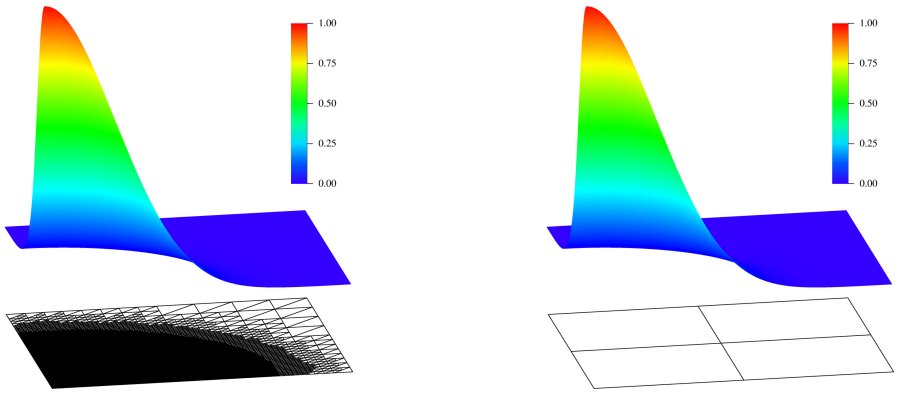


Figure 10: Solution of the Poisson equation on a non-conform refined hexahedral grid.



(a) non-conforming adaptively refined triangular grid (`ALUSimplexGrid`) and P_1 Lagrange elements.

(b) Cartesian grid (`YaspGrid`) and P_6 Lagrange elements.

Figure 11: Solution of Poisson's equation. the P_1 solution 105338 grid cells were necessary while for the P_6 solution only 4 cells. In both cases the L^2-error is approximately $2.8 \cdot 10^{-5}$.

In Figure 12 the run times for solving this problem on a very fine uniformly refined grid containing about $1.34 \cdot 10^8$ hexahedrons using different grid implementations and different numbers of processors is shown. Each computation was done twice using different methods of communication. These communication methods are described in Section 5.1. One communication method uses the data communication provided by the DUNE grid interface, which is implemented by the class `DefaultCommunicationManager`, see Section 5.1. The other method builds a cache holding information of all DoFs that need to be exchanged during a communication procedure and thus a grid traversal is not necessary and the message buffers can be allocated at once since the size is already known. The implementing class is `CachedCommunicationManager`, see Section 5.1. The computations using cached communication are tagged with **cached** in both plots of Figure 12. In the left part of Figure 12 one can see that the computation using `ALUCubeGrid` and the cached communication is more than twice as fast as the run using the non-cached communication. With a higher number of processors the gap becomes even larger. For `YaspGrid` (a Cartesian grid) the cached communication is only slightly faster since the traversal of grid cells is already very cheap. Also, the results for the computation using the cached communication are more uniformly decreasing. `YaspGrid` was used without overlapping cells and for `ALUCubeGrid` all vertices with partition type `ghost` are neglected as unknowns for the solution, right hand side, and the system matrix. Comparing both grids, we can see from Figure 12 (left) that they show almost identical run times. We state that the use of the cached communication is essential for an efficient parallel communication using a large number of communication operations during execution of the numerical solution algorithm such as a linear solver or also an ODE solver does.

7.1.2 The Euler Equations

For a compressible inviscid fluid the Euler equations of gas dynamics have the following form:

$$\partial_t \boldsymbol{u} + \sum_{j=1}^{d} \partial_{x_j} \boldsymbol{f}_j(\boldsymbol{u}) = 0, \qquad \text{in } ([0,T) \times \Omega \subset \mathbb{R}^d), \quad d \in \{1,2,3\} \tag{12}$$

where the vector of the conservative variables is

$$\boldsymbol{u} = (\rho, \rho\boldsymbol{v}, \rho E)^T, \quad \rho\boldsymbol{v} = (\rho v_1, \ldots, \rho v_d)^T,$$

where E the total energy. We assume that solutions $\boldsymbol{u}$ of 12 take its values in the set of states

$$\Psi := \left\{(\rho, \rho\boldsymbol{v}, \rho E) \mid \rho > 0, \boldsymbol{v} \in \mathbb{R}^d, \rho E - \frac{\rho}{2}|\boldsymbol{v}|^2 > 0\right\},$$

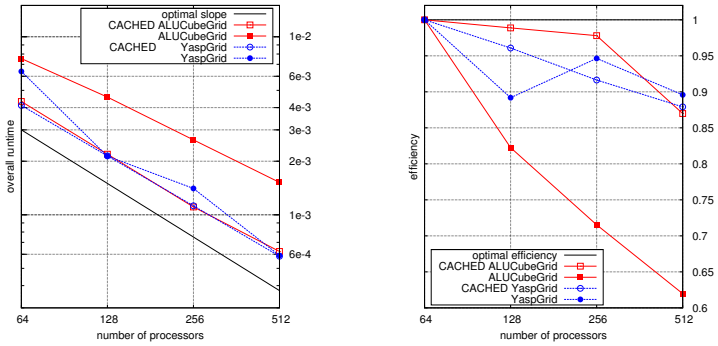


Figure 12: Run times (right) and efficiency (left) for a parallel computation for the Poisson equation. The plots show the results for two different grid implementation (`ALUCubeGrid` and `YaspGrid`) on a hexahedral grid containing about $1.34 \cdot 10^8$ elements and the problem size is about $1.35 \cdot 10^8$ unknowns.

and the convective flux functions for $i = 1, ..., d$:

$$\boldsymbol{f}_i(\boldsymbol{u}) := \left(\rho v_i, \rho v_i \boldsymbol{v} + P(\boldsymbol{u})\boldsymbol{e}_i, v_i(\rho E + P(\boldsymbol{u}))\right)^T$$

where $\boldsymbol{e}_i$ is the unit vector in direction i. The system is closed by the equation of state for an *ideal gas* where the pressure is given by $P(\boldsymbol{u}) = (\gamma - 1)\left[\rho E - \frac{\rho}{2}|\boldsymbol{v}|^2\right]$, where γ is the adiabatic constant (see for example [29]).

Discretization

Now, to rewrite the system in the form of equation (10) we simply have to define the operator $\mathcal{L}_{\text{expl,h}}$. The operator

$$\mathcal{L}_{\text{expl,h}}[\boldsymbol{u}] := -\nabla \cdot \boldsymbol{F}(\boldsymbol{u})$$

only contains the conservative part with $\boldsymbol{F}(\boldsymbol{u}) := (\boldsymbol{f}_1, \ldots, \boldsymbol{f}_d)$.
With the definition of $\mathcal{L}_{\text{expl,h}}$ and an appropriate numerical flux the spatial discretization is already obtained. Appropriate numerical flux functions are for example the Local-Lax-Friedrichs flux or the HLL flux function which can be found in standard textbooks on the subject (see for example [29, 31]).
We apply the Runge-Kutta Discontinuous Galerkin discretization for this problem. The discretization includes a limiter based stabilization technique. Both is described in detail in [16].

Implementation details

The discontinuous Galerkin space is implemented by the class `DiscontinuousGalerkinSpace` (see Section 4.2.5) providing L^2 orthonormal basis functions. For time discretization a Runge Kutta solver has been used. This solver is implemented by the class `ExplicitRungKuttaSolver` (see Section 4.4). Adaptation and load balancing is handled by the class `AdaptationManager` (see Section 5.2). During each sub step of the ODE solver an `interior-ghost` (see Example 65) communication has to be performed. This is done using the cached communication implemented by the class `CachedCommunicationManager` described in Section 5.1.

Numerical Results

As benchmark problem we consider the *Forward Facing Step* (see [16] for details) for $d = 3$.
In Figure 13 the density distribution including the adaptively refined grid can be found. One can also see the underlying grid partitioning. The simulation has been done on the parallel super computer XC4000 of the SSC Karlsruhe using 512 processors. Quadratic polynomial basis

functions have been used. The final grid contains about 4.5 million grid cells which leads for this example to about $2.25 \cdot 10^8$ unknowns. Grid adaptation is performed in each time step. If the local grid adaptation leads to an unbalance of work load then a dynamic load balancing is performed such that the work is again equally distributed between the processors[2] again. A detailed EOC analysis of this discretization as well as other benchmark problems for this problem can be found in [16].

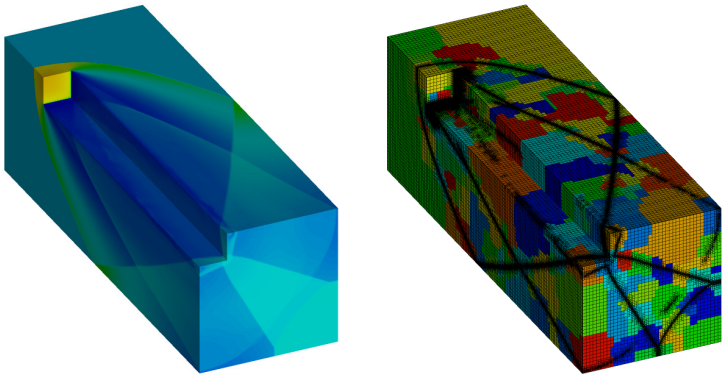


Figure 13: Density distribution, adapted grid and partitioning of the grid at $T = 2$. The calculation used `ALUCubeGrid` and 512 processors. Quadratic basis functions have been used.

In Figure 14 the parallel performance of the code is presented. In the left part the run times for one time step as well as only for the ODE solver are plotted for the runs on 128, 256, and 512 processors. In the right part the efficiency due to this run times is shown. One can see that the overall efficiency is above 0.93 and the efficiency of the serial part (ODE solver) of the code is above 0.96 which is very close to the optimal value of 1. This indicates that the parallelization of the code is very efficient for this problem.

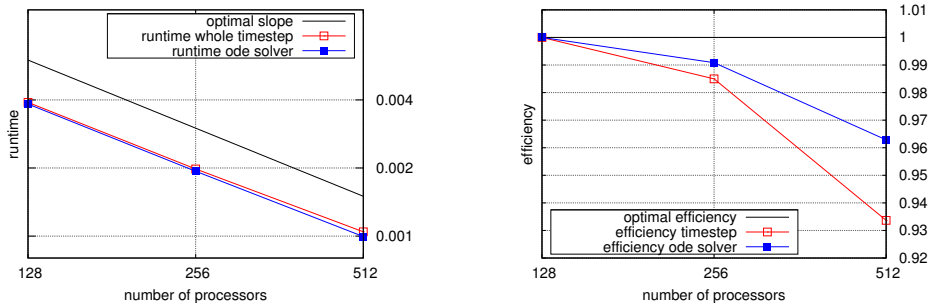


Figure 14: Run times for a parallel computation (left) and efficiency of the parallel code (right) for the Euler equations using the third order stabilized DG discretization using hexahedral elements (`ALUCubeGrid`). On the left plot the runtime for one complete timestep (all) and the runtime of one timestep taken by the ODE solver (ode) are shown. In the right plot the efficiency of the code is shown.

7.1.3 The Stokes problem

As a third benchmark problem we consider a local discontinous Galerkin approximation of the Stokes equation. Therefore, we look at the following Stokes system on a bounded domain $\Omega \subset \mathbb{R}^d$, $d = 2, 3$ with Dirichlet data.

[2] The load balancing considers the number of grid elements considered in the numerical algorithm and is based on the graph partitioning algorithm provided by METIS [38].

$$-\Delta \boldsymbol{u} + \nabla p = f \quad \text{in} \quad \Omega, \tag{13}$$
$$\nabla \cdot \boldsymbol{u} = 0 \quad \text{in} \quad \Omega, \tag{14}$$
$$\boldsymbol{u} = g_{\mathcal{D}} \quad \text{on} \quad \partial\Omega \tag{15}$$

with the compatibility condition

$$\int_{\partial\Omega} g_{\mathcal{D}} \cdot \mathbf{n} = 0. \tag{16}$$

Here $\boldsymbol{u} : \Omega \to \mathbb{R}^d$ denotes the velocity, and $p : \Omega \to \mathbb{R}^d$ the pressure. $f : \Omega \to \mathbb{R}^d$ is a given vector field and $g_{\mathcal{D}} : \partial\Omega \to \mathbb{R}^d$ given boundary data.

Discretization

The Stokes system is discretized using a locally conservative version of the local discontinuous Galerkin method introduced in [12]. For this method the Stokes equations are first written as a System of first order partial differential equations with the additional unknown $\sigma = \nabla u$.

Implementation details

The discontinuous Galerkin space is implemented by the class `DiscontinuousGalerkinSpace` (see Section 4.2.5) providing L^2 orthonormal basis functions up to order 5. The operators of the resulting first order system for the unkowns $\sigma, \boldsymbol{u}, p$ are realized via the concept of discrete combined operators and passes (see definition 37). For further details we refer to [28].

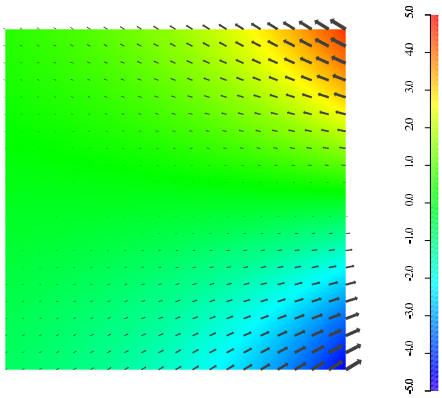


Figure 15: Pressure (color coded) and velocity field (arrows) for the benchmark problem of the Stokes system.

Numerical results

As a benchmark problem we look at the Stokes system on $\Omega = (-1,-1)^2$ where the data $f, g_{\mathcal{D}}$ are chosen in such a way, that the exakt solution (u, p) is given as

$$\begin{aligned} u_1(x_1, x_2) &= -e^{x_1}(x_2 \cos x_2 + \sin x_2), \\ u_2(x_1, x_2) &= e^{x_1} x_2 \sin x_2, \\ p(x_1, x_2) &= 2e^{x_1} \sin x_2. \end{aligned}$$

The stabilization parameters in the local discontinuous Galerkin scheme were chosen as $C_{11} = h^{-1}, D_{11} = h, \mathbf{C}_{12} = \mathbf{D}_{12} = 0$. The numerical results are taken from the Diploma thesis of [28]. Table 3 shows the numerical error and the experimental order of convergence (EOC) in the case, where $\sigma, \boldsymbol{u}$ and p are taken as local polynomials of the same order. A plot of the numerical results for polynomial order 1 on a triangular grid with 8192 elements is shown in Figure 15.

Table 3: Convergence study for the LDG approximation with local polynomial order 2 and 3.

Size	$\|\|u-u_h\|\|_2$	EOC	$\|\|p-p_h\|\|_2$	EOC
128	$4.998e-4$	3.23	$4.781e-3$	2.27
512	$6.282e-5$	3.26	$1.134e-3$	2.07
2048	$7.889e-6$	3.27	$2.810e-4$	2.01
8192	$1.150e-6$	3.06	$7.638e-5$	1.87

Size	$\|\|u-u_h\|\|_2$	EOC	$\|\|p-p_h\|\|_2$	EOC
8	$3.159e-3$	3.85	$1.797e-2$	3.28
32	$2.179e-4$	4.30	$2.782e-3$	2.69
128	$1.370e-5$	4.31	$3.584e-4$	2.95
512	$9.010e-7$	4.26	$4.548e-5$	2.97

7.2 Advanced applications

7.2.1 Free surface shallow water flow

Problem formulation

We solve the 3D free surface Navier-Stokes equations with a hydrostatic pressure assumptions, a model suitable for shallow flows. Taking into account the orography b and denoting with h the free surface, the computational domain is

$$\Omega(t) = \left\{(\boldsymbol{x}, z)^T \in \mathbb{R}^d \,:\, \boldsymbol{x} \in \Omega_{\boldsymbol{x}},\, b(\boldsymbol{x}) < z < b(\boldsymbol{x}) + h(\boldsymbol{x}, t)\right\}.$$

where $\Omega_{\boldsymbol{x}} \subset \mathbb{R}^{d-1}$ is a fixed domain over which b and $h(\cdot, t)$ can be represented as functions. The following system for the 3d velocity field $\mathbf{u} = (\mathbf{u}_{\boldsymbol{x}}, w)^T$ results from a *Shallow Water Scaling* of the incompressible Navier-Stokes equations as it is presented in [21]:

$$\begin{aligned} \partial_t h + \nabla_{\boldsymbol{x}} \cdot \left(\int_b^{b+h} \mathbf{u}_{\boldsymbol{x}} dz\right) &= 0 && \text{in } \Omega_{\boldsymbol{x}}, \\ \partial_t \mathbf{u}_{\boldsymbol{x}} + (\mathbf{u}\cdot\nabla)\mathbf{u}_{\boldsymbol{x}} + g\nabla_{\boldsymbol{x}} h &= -g\nabla_{\boldsymbol{x}} b + \partial_z\left(\mu \partial_z \mathbf{u}_{\boldsymbol{x}}\right) && \text{in } \Omega(t), \\ \nabla\cdot\mathbf{u} &= 0 && \text{in } \Omega(t), \end{aligned} \tag{17}$$

where $g > 0$ is the gravitational constant. For the full set of boundary conditions associated with this system we refer to [22].

Discretization

The system is solved using a local discontinuous Galerkin approach. Using a sigma transformation the problem is represented on a fixed domain $\omega \times [0, 1]$ where $\omega \subset \mathbb{R}^2$.

Implementation details

The method uses the concept of combined operators (see Definition 37), employing three passes: $(h, \mathbf{u}_{\boldsymbol{x}})$:

1. compute the integrals of the horizontal velocities $\int_b^{b+h} \mathbf{u}_{\boldsymbol{x}} dz$
2. compute the vertical velocity based on the divergence constraint: $\partial_z w = -\nabla_{\boldsymbol{x}} \cdot \mathbf{u}_{\boldsymbol{x}}$
3. compute the advection-diffusion terms in (17)

As discrete function space the `DiscontinuousGalerkinSpace` (see Section 4.2.5) is used. This spatial discretization is combined with an implicit-explicit Runge-Kutta solver treating the diffusion terms implicitly to increase the stability of the method (see `SemiImplicitOdeSolver` in Section 4.4).
To increase the efficiency of the first two steps of the algorithm a special semi-discrete prism grid (Figure 16) is used, the grid is structured in the z direction so that the computation of the vertical integral and transport is easy to compute.

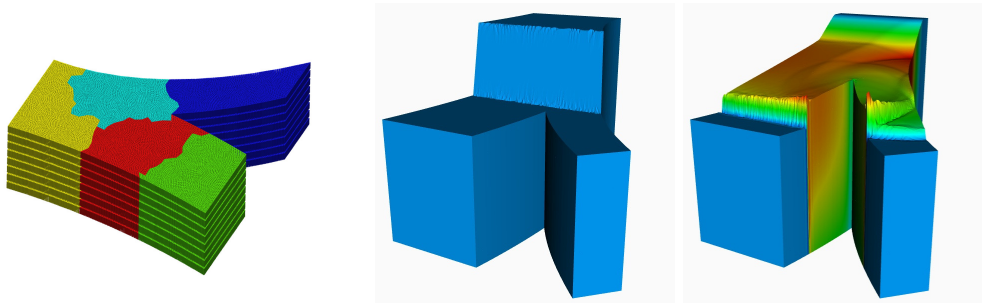


Figure 16: (*left*) partitioning of a 3d semi-structured prism grid (*middle*) 3d representation of initial conditions (*right*) solution to a latter time

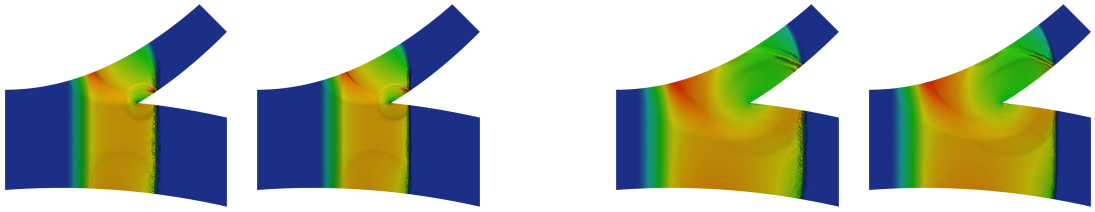


Figure 17: Two points in time of a simulation using piecewise linear basis functions on coarse grid (917760 elements) (left) and on finer grid (3671040 elements) (right)

Numerical results

Figure 17 shows an example computation of a wave moving from left to right hitting a fork in the river. The convergence of the scheme is demonstrated showing results on two different grids for two time steps.

7.2.2 Registration of medical images

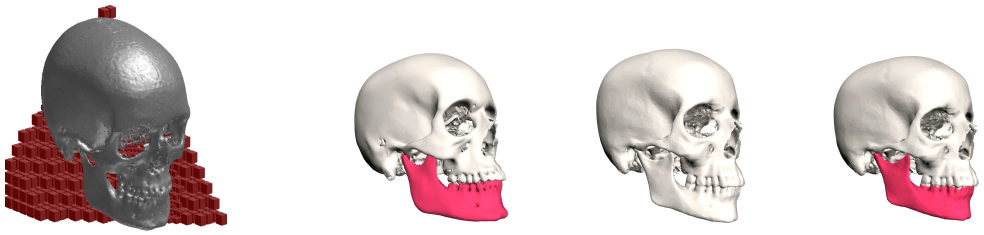
Problem formulation

We consider the problem of non-rigid, point-to-point registration of two 3D surfaces. In the example of medical imaging these might represent two femur or human skulls. The goal is to construct a statistical model from a large set of images.

To avoid restrictions on the topology, we represent the surfaces as a level-set of their signed distance function, $I0$ and $I1$, respectively. Correspondence is established by finding a displacement field that minimizes the sum of squared difference between the function values as well as their mean curvature, denoted in the following by $H0$ and $H1$. This leads to following (regularized) minimization problem: $\mathcal{J}[u] = \mathcal{D}[u] + \beta\mathcal{C}[u] + \alpha\mathcal{R}[u]$ with

$$\begin{aligned}
\mathcal{D}[u] &= \frac{1}{2}\int_\Omega \tfrac{1}{Q_I(x)}\left(I_0(x+u(x)) - I_1(x)\right)^2 dx \ , \\
\mathcal{C}[u] &= \frac{1}{2}\int_\Omega \tfrac{1}{Q_H(x)}\left(H_0(x+u(x)) - H_1(x)^2\right) dx \ , \\
\mathcal{R}[u] &= \frac{1}{2}\sum_{l=1}^{3}\int_\Omega |\nabla u_l|^2 \, dx \ ,
\end{aligned}$$

using $Q_F(x) = |\nabla F_0(x)|^2 + (F_0(x) - F_1(x))^2$.



(a) Grid structure (b) reference (c) targed (d) registered

Figure 18: A labeling defind on a reference skull (b) is automatically transformed to a target skull (c) producing (d) using the vector field u. The adaptive grid used to compute u is shown in (a).

Discretization

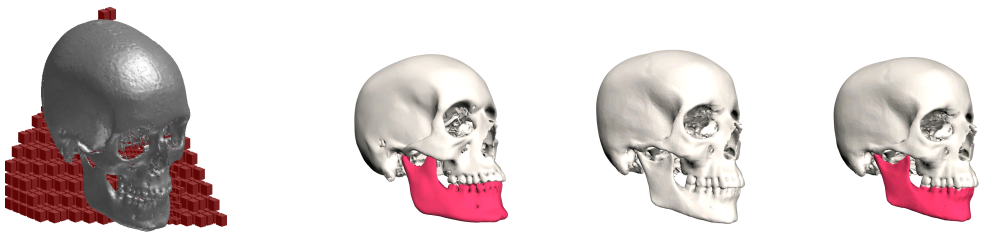
We use the local discontinuous Galerkin method to solve the Euler-Lagrange equation corresponding to the minimization problem. These are a system of non-linear elliptic equations of the form: $-\Delta u = S(u)$

Implementation details

We use a pseudo time stepping scheme to obtain the solution to the elliptic problem using a semi-implicit Runge-Kutta method. The Laplace term is treated implicitly and the non-linear source term explicitly. The process is initiated using a coarse grid and after a fixed number of pseudo time steps the grid is locally refined around the zero level set of $I0$. This iteration process is repeated and the grid successivly refined. We use the `DiscontinuousGalerkinSpace` (see Section 4.2.5) and the implicit-explicit Runge-Kutta solver `SemiImplicitOdeSolver` (see Section 4.4).

Numerical results

Figure 19 shows the registration of two skulls. The reference image ($I0$), the target ($I1$) and the warped image ($I0(x + u(x))$) are shown. To demonstrate the effectivness of the algorithm the mandible has been labeled on the reference image and the computed vector field u is used to transform this label to the targed. More details can be found in [17].



(a) Grid structure (b) reference (c) targed (d) registered

Figure 19: A labelling defind on a reference skull (a) is automatically transformed to a target skull (b) producing (c) using the vector field u.

8 Acknowledgments

This paper documents the design principles that were used for the implementation of the DUNE-FEM module. The authors would like to thank all contributors who helped with ideas, implementations and testing. In particular, these are S. Brdar, A. Burri, M. Drohmann, M. Droske, C. Gersbacher, B. Haasdonk, P. Henning, K. Hermsdörfer, N. Jung, M. Kränkel, E. Louw, T. Malkmus, K. Michel, T. Müller, N. Shokina, C. Tillmanns, and U. Schweizer, The first author was supported by the Landesstiftung Baden-Württemberg. The second author was supported by the Federal Ministry of Economics and Technology of Germany (BMBF) under contract number 03SF0310C.

A Detailed description of classes in DUNE-FEM

In the appendix we give a detailed description of the central classes used in the DUNE-FEM template library. For every object described in Section 2 there will be an interface class translating the mathematical functionality into an object using the programming language C++. In the following we distinguish between static parameters of a class or method, called template parameters in C++, and dynamic parameters. As static parameters are known at compile time, the compiler can use this information to improve the generated code. For each interface class there is an enumeration of the most important features of the class starting with the template parameters, followed by a list of exported types, and finished by the method list of the class. Dynamic construction parameters are described only for the objects implementing a certain interface.

A.1 Subsets of hierarchic grids

Class 1 (`IndexSet`)

An `IndexSet` represents the interface for an index set $\Lambda_\mathcal{G}$ (see Definition 11).

Template Parameters	
`GridPart`	type of grid part representing $\mathcal{G}$

Interface Methods	
`int size(`$\hat{E}$`)`	returns the size of the index set, i.e., $s^{\Lambda_\mathcal{G}^{\hat{E}}}$
`int index(`E`)`	returns $\lambda_\mathcal{G}(E)$, the index of the entity E
`int subIndex<c>(`E`,`l`)`	returns $\lambda_\mathcal{G}(E^{c,l})$, where $E^{c,l}$ is the l-th subentity with codimension c of element E
`bool contains(`E`)`	returns `true` if $\Lambda_\mathcal{G}$ provides an index for E, i.e., if $E \in \mathcal{G}$, `false` otherwise

Class 1 (`IndexSet`)

Class 2 (`PersistentIndexSet` $\longrightarrow$ `IndexSet`)

The `PersistentIndexSet` represents the interface for a persistent index set $\Lambda_\mathcal{G}$ (see Definition 46). This interface class automatically adds itself to the `DofManager`'s list of persistent index sets (see Class 30 and Class 34).

Template Parameters	
`GridPart`	type of grid part representing $\mathcal{G}$

Interface Methods	
`void insertEntity(`E`)`	create an index for the entity E and all its subentities
`void removeEntity(`E`)`	mark the index for E as unused
`void resize()`	insert all new elements on a previously adapted grid, making the index set consistent with the new state

Class 2 (`PersistentIndexSet`)

Class 3 (`ConsecutivePersistentIndexSet` $\longrightarrow$ `PersistentIndexSet`)

A `ConsecutivePersistentIndexSet` represents the interface for a consecutive and persistent index set $\Lambda_\mathcal{G}$ (see Definition 48). In particular, this class provides methods `numberOfHoles`, `oldIndex`, and `newIndex` implementing the mapping $\xi_\mathcal{G}^n$ (see Definition 48).

Template Parameters	
`GridPart`	type of grid part representing $\mathcal{G}$

Interface Methods	
`bool compress()`	make the index set consecutive according to Definition 55 and return `true` if $s_h > 0$, i.e., if the number of holes was non-zero
`int numberOfHoles()`	return the number of holes s_h that existed before the last compression
`int oldIndex(`i`)`	return the old index for hole i, i.e., $\lambda_{\mathcal{G}}^{old}(i)$, $0 \leq i < s_h$
`int newIndex(`i`)`	return the new index for hole i, i.e., $\lambda_{\mathcal{G}}^{new}(i)$, $0 \leq i < s_h$

Class 3 (`ConsecutivePersistentIndexSet`)

Class 4 (`GridPart`)

A `GridPart` describes the interface for a grid $\mathcal{G}$ (see Definition 4) equipped with a given index set.

Template Parameters	
`Traits`	traits class from which the types are extracted

Exported Types	
`GridType`	type of hierarchical grid $\mathcal{H}$ from which a subset of entities is selected
`IndexSetType`	type of index set returned by method indexSet
`Codim<c>::IteratorType`	type of iterator returned by methods begin/end for codimension `c`
`IntersectionIteratorType`	type of intersection iterator returned by methods ibegin/iend

Interface Methods	
`GridType& grid()`	returns a reference to the hierarchic grid $\mathcal{H}$
`IndexSetType& indexSet()`	returns a reference to the index set $\Lambda_{\mathcal{G}}$
`Codim<c>::IteratorType begin/end<c>()`	returns an iterator pair to iterate the half-open interval $[begin, end)$ which contains all elements of the set $\mathcal{E}^c$
`IntersectionIteratorType ibegin/iend(`E`)`	returns an iterator pair to iterate over all intersections of E with other codimension 0 entities
`void communicate(`$dH, if\ \ dir$`)`	communicate data on $\mathcal{G}$ using the `DataHandle` dH on the subset of entities selected by the communication interface if and the communication direction dir

Class 4 (`GridPart`)

A.2 Discrete functions

Class 5 (`FunctionSpaceInterface`)

The `FunctionSpaceInterface` represents the interface for a function space $V^{\Omega,U}$ (see Definition 16).

Template Parameters	
`Traits`	traits class from which the types are extracted

Exported Types and Constants	
`int dimDomain`	$dim(\Omega_{\mathcal{G}})$

Exported Types and Constants	
`int dimRange`	$dim(U)$
`DomainFieldType`	C++ type modeling the field of $\Omega_{\mathcal{G}}$, e.g., `double`
`RangeFieldType`	C++ type modeling the field of U, e.g., `double`
`DomainType`	C++ type modeling the elements of $\Omega_{\mathcal{G}}$, e.g., `FieldVector< DomainField,dimDomain >`
`RangeType`	C++ type modeling the elements of U, e.g., `FieldVector< RangeField,dimRange >`
`JacobianRangeType`	C++ type modeling the Jacobian matrix, e.g., `FieldMatrix< RangeField,dimDomain,dimRange >`
`HessianRangeType`	type of the Hessian of the function, e.g., `FieldVector< JacobianRangeType, dimRange >`
`ScalarFunctionSpaceType`	type of the corresponding scalar function space, i.e., this function space with `dimRange` $= 1$

Class 5 (`FunctionSpaceInterface`)

Class 6 (`Function`)

A `Function` represents the interface for a function $v \in V^{\Omega,U}$.

Template Parameters	
`FunctionSpace`	type of function space $V^{\Omega,U}$

Exported Types and Constants	
	all exported types and constants from `FunctionSpace` are forwarded
`FunctionSpaceType`	type of function space $V^{\Omega,U}$

Interface Methods	
`void evaluate(`$\boldsymbol{x}$`,`val`)`	evaluates $val = v(\boldsymbol{x})$ for $\boldsymbol{x} \in \Omega$
`FunctionSpaceType& space()`	returns a reference to the function space $V^{\Omega,U}$

Class 6 (`Function`)

Class 7 (`LocalFunction`)

A `LocalFunction` represents the interface for a local function v_E of a function $v \in V^{\Omega_{\mathcal{G}},U}$ on an element E (see Definition 17).

Template Parameters	
`FunctionSpace`	type of function space $V^{\Omega_{\mathcal{G}},U}$

Exported Types and Constants	
	all exported types and Constants from `FunctionSpace` are forwarded

Interface Methods	
`void evaluate(`$\hat{\lambda}$`,`val`)`	evaluates the v_E in $\hat{\lambda} \in \hat{E}$, i.e., $val = v_E(F_E(\hat{\lambda}))$

Class 7 (`LocalFunction`)

Class 8 (`GridFunction` $\longrightarrow$ `Function`)

A `GridFunction` represents the interface for a grid function $v \in V^{\Omega_{\mathcal{G}},U}$ (see Definition 17).

Template Parameters	
`FunctionSpace`	type of function space $V^{\Omega_{\mathcal{G}},U}$

Additional Exported Types	
`LocalFunctionType`	type of local function v_E

Additional Interface Methods	
`LocalFunctionType localFunction(`E`)`	returns a `LocalFunction` object v_E for element E

Class 8 (`GridFunction`)

Class 9 (`BaseFunction`)

The class `BaseFunction` describes the interface for a single base function.

Template Parameters	
`FunctionSpace`	type of $V^{\Omega_\mathcal{G},U}$

Exported Types and Constants	
	all exported types and constants from `FunctionSpace` are forwarded
`FunctionSpaceType`	type of $V^{\Omega_\mathcal{G},U}$

Virtual Interface Methods	
`virtual void evaluate(`$v[0]$`,`x`,`val`)`	v has length 0, so the base function is evaluated, i.e., $val = \varphi(x)$
`virtual void evaluate(`$v[1]$`,`x`,`val`)`	v has length 1, so the first derivative in direction v_0 of the base function is evaluated, i.e., $val = \partial_{v_0}\varphi(x)$
`virtual void evaluate(`$v[2]$`,`x`,`val`)`	v has length 2, so the second derivative in directions v_0, v_1 is evaluated, i.e., $val = \partial_{v_0}\partial_{v_1}\varphi(x)$

Class 9 (`BaseFunction`)

Class 10 (`BaseFunctionSet`)

A `BaseFunctionSet` represents the interface for a base function set $\mathcal{B}_E$ (see Definition 18).

Template Parameters	
`FunctionSpace`	type of $V^{\Omega_\mathcal{G},U}$

Exported Types and Constants	
	all types and constants from `FunctionSpace` are forwarded
`FunctionSpaceType`	type of $V^{\Omega_\mathcal{G},U}$

Interface Methods	
`int numBaseFunctions()`	returns the number of base functions, i.e., $\lvert\mathcal{B}_E\rvert$
`void evaluate(`i`,`$\hat{\lambda}$`,`val`)`	evaluates the base function φ_i in $\hat{\lambda} \in \hat{E}$, i.e., $val = \varphi_i(\hat{\lambda})$
`void jacobian(`i`,`$\hat{\lambda}$`,`val`)`	evaluates the Jacobian of the base function φ_i in $\hat{\lambda} \in \hat{E}$, i.e., $val = \mathrm{D}\varphi_i(\hat{\lambda})$
`void hessian(`i`,`$\hat{\lambda}$`,`val`)`	evaluates the Hessian of the base function φ_i in in $\hat{\lambda} \in \hat{E}$, i.e., $val = \mathrm{D}^2\varphi_i(\hat{\lambda})$

Class 10 (`BaseFunctionSet`)

Class 11 (`EvaluationPointList`)

An `EvaluationPointList` represents the interface for a set of points $P_{\hat{E}}$ located in the reference element $\hat{E}$. It is needed for the evaluation of base functions and similar operations.

Template Parameters	
`GridPart`	type of `GridPart` representing the grid $\mathcal{G}$ (needed to determine the type of the `Intersection`)
`int codim`	$dim(\Omega_{\mathcal{G}}) - dim(\hat{E})$ (currently codimension 0 and 1 are supported)

Exported Types and Constants	
`FieldType`	C++ type modeling the field of $\hat{E}$
`int dimension`	$dim(\hat{E})$
`CoordinateType`	type of a point $\hat{\lambda} \in \hat{E}$, e.g., `FieldVector<FieldType,dimension>`
`LocalCoordinateType`	type of local evaluation points (for codimension 1 only)

Interface Methods	
`int nop()`	returns the number of points in the set, i.e., $\lvert P_{\hat{E}} \rvert$
`CoordinateType& point(`i`)`	returns $\hat{\lambda}_i$, the coordinates for point i
`LocalCoordinateType& localPoint(`i`)`	returns the local coordinates for point i, (for a face point list, the coordinates are local with respect to the face)
`QuadraturePointWrapperType operator[](`i`)`	returns the evaluation point $(\hat{\lambda}_i, i)$, required for specialization of base function evaluations in terms of caching
`int order()`	returns order k for which a quadrature based on $P_{\hat{E}}$ would be exact
`size_t id()`	returns a unique id for $P_{\hat{E}}$, needed to identify cache lines

Class 11 (`EvaluationPointList`)

Class 12 (`ElementQuadrature` $\longrightarrow$ `EvaluationPointList`)

The class `ElementQuadrature` inherits the class `EvaluationPointList` and represents the interface for a quadrature $Q_{\hat{E}}$
Note that this quadrature interface differs slightly from the interface presented in DUNE-GRID.

Template Parameters	
	see `EvaluationPointList` (Class 11)

Constructors for codimension 0	
`ElementQuadrature(`E`,`k`)`	create a quadrature (codimension $c = 0$) exact up to order k an element E

Constructors for codimension 1	
`ElementQuadrature(`$\mathcal{G}$`,`e`,`k`,flag)`	create a face quadrature (codimension $c = 1$) exact up to order k for an intersection e; the coordinates are given with respect to the inside entity (`flag = INSIDE`) or the outside entity (`flag = OUTSIDE`)

Additional Interface Methods	
`FieldType weight(`i`)`	returns the weight ω_i for point $\hat{\lambda}_i$

Class 12 (`ElementQuadrature`)

Class 13 (`CachingPointList` $\longrightarrow$ `EvaluationPointList`)

A `CachingPointList` represents a list of points for evaluation of base functions implementing

the interface of `EvaluationPointList` using a caching mechanism. Caching can only be used in combination with a `CachingStorage` (see Class 15).

Template Parameters	
	see `EvaluationPointList` (Class 11)

Additional Interface Methods	
`int cachingPoint(`i`)`	returns the number of the caching point of point i, which might be twisted according to a face twist

Class 13 (`CachingPointList`)

Class 14 (`CachingQuadrature` $\longrightarrow$ `CachingPointList, ElementQuadrature`)

A `CachingQuadrature` represents a quadrature implementing the `CachingPointList`, thus providing caching for base function evaluations and the interface of a `ElementQuadrature` for numerical integration. Caching can only be used in combination with a `CachingStorage` (see Class 15).

Template Parameters	
	see `EvaluationPointList` (Class 11)

Constructors	
	see `ElementQuadrature` (Class 12)

Class 14 (`CachingQuadrature`)

Class 15 (`CachingStorage`)

A `CachingStorage` implements the `BaseFunctionSet` interface. The base functions are evaluated only once and the values are stored in a look-up table for later use. Evaluations of base functions in arbitrary points are forwarded to the real implementation of the base function.

Template Parameters	
`FunctionSpace`	type of $V^{\Omega_{\mathcal{G}},U}$

Exported Types and Constants	
	all exported types and constants from `FunctionSpace` are forwarded

Constructors	
`CachingStorage(`f`)`	Creates a `CachingStorage`, creating the base function objects by use the base function factory f

Class 15 (`CachingStorage`)

Class 16 (`DofMapper`)

The `DofMapper` represents the interface for a DoF mapper $\mu_{\mathcal{G}}$ (see Definition 18).

Template Parameters	
`Traits`	traits class from which the types are extracted

Exported Types	
`EntityType`	type of (codimension 0) entity E for which the mapping is provided
`DofMapIteratorType`	type of iterator for the local to global DoF map returned by begin/end

Interface Methods	
`int size()`	returns the number of global DoFs, i.e., $\lvert\mathcal{B}_{\mathcal{G}}\rvert$

Interface Methods	
`int maxNumDofs()`	returns the maximal number of local DoFs on an entity
`int numDofs(`E`)`	returns the number of local DoFs on entity E, i.e., $\lvert\mathcal{B}_E\rvert$
`int mapToGlobal(`E`,`l`)`	returns the global index of the local DoF l on codimension 0 entity E, i.e., $\mu_E(l)$; this includes local DoFs associated with subentities
`DofMapIteratorType begin/end(`E`)`	returns an iterator pair for the local to global DoF map $\mu_E(0), \ldots, \mu_E(\lvert\mathcal{B}_E\rvert - 1)$
`int mapEntityDofToGlobal(`E`,`l`)`	returns the global index of the l-th DoF associated with entity E of arbitrary codimension; in contrast to `mapToGlobal`, this mapping only includes DoFs associated with E

Interface Method for use in ManagedDofStorage (see Class 33)	
`int numBlocks()`	returns the number of separate blocks in the DoF vectors
`int numberOfHoles(`b`)`	returns the number of holes in block b, i.e., n_h^b
`int oldIndex(`b`,`k`)`	returns the old index for hole k in block b, i.e., $\mu_{\mathcal{G}}^{old}(b,k)$, $0 \leq k < n_h^b$
`int newIndex(`b`,`k`)`	returns the new index for hole k in block b, i.e., $\mu_{\mathcal{G}}^{new}(b,k)$, $0 \leq k < n_h^b$
`void update(oversize)`	update insertion points for blocks[3]; if `oversize` is `true`, the size is overestimated to avoid unnecessary memory movement; this method is used in the `resize` and `dofCompress` methods of a `ManagedDofStorage` (see Class 33)
`int oldOffSet(`b`)`	returns the old offset (offsets change when `update` is called) of block b
`int offSet(`b`)`	returns current offset of block b

Class 16 (`DofMapper`)

Class 17 (`DiscreteFunctionSpace` $\longrightarrow$ `FunctionSpace`)

A `DiscreteFunctionSpace` represents the interface for a discrete function space $\mathcal{D}_{\mathcal{G}}^V$ (see Definition 18).

Template Parameters	
`Traits`	traits class from which the types are extracted

Additional Exported Types and Constants	
`FunctionSpaceType`	type of function space $V^{\Omega_{\mathcal{G}},U}$
`GridPartType`	type of grid part representing $\mathcal{G}$
`IndexSetType`	type of index set $\Lambda_{\mathcal{G}}$
`MapperType`	type of DoF mapper $\mu_{\mathcal{G}}$
`int polynomialOrder`	polynomial order of the base functions

Interface Methods	
`GridPartType& gridPart()`	returns a reference to the grid part representing $\mathcal{G}$
`IndexSetType& indexSet()`	returns a reference to the index set $\Lambda_{\mathcal{G}}$

[3] This is only necessary for DoF mappers with more than one block such as DoF mappers for higher order Lagrange spaces or hybrid grids.

Interface Methods	
`MapperType& mapper()`	returns a reference to the DoF mapper $\mu_{\mathcal{G}}$
`IteratorType& begin/end()`	returns a pair of iterators to traverse the half-open interval $[begin, end)$ containing all elements of $\mathcal{G}$
`BaseFunctionSetType baseFunctionSet(`E`)`	returns the base function set $\mathcal{B}_E$
`int order()`	returns polynomial order of the discrete function space, if available, otherwise -1
`void communicate(`$u_{\mathcal{G}}$`)`	communicates data for the discrete function $u_{\mathcal{G}}$ with the appropriate communication interface, direction, and operation, e.g., `+=` or `=`, on the data
`void communicate(`$u_{\mathcal{G}}$`,`γ`)`	communicates data for the discrete function $u_{\mathcal{G}}$ with the appropriate communication interface and direction; the user-defined operation γ is used to combine the data
`InterfaceType communicationInterface()`	returns the default communication interface for this discrete function space
`CommunicationDirection communicationDirection()`	returns the default communication direction for this discrete function space

Class 17 (`DiscreteFunctionSpace`)

Class 18 (`LagrangeDiscreteFunctionSpace` $\longrightarrow$ `DiscreteFunctionSpace`)

This class implements the Lagrange discrete function space described in Definition 26. Note that the basic structure for all implemented discrete function spaces is exactly the same and their description is therefore skipped.

Template Parameters	
`FunctionSpace`	type of $V^{\Omega_{\mathcal{G}},U}$
`GridPart`	type of grid $\mathcal{G}$
`int polynomialOrder`	polynomial order of the basis functions
`Storage`	base function storage, which can be either `CachingStorage` (which is the default value) or `SimpleStorage`

Constructors	
`LagrangeDiscreteFunctionSpace (`$\mathcal{G}$`)`	creates a Lagrange discrete function space on $\mathcal{G}$; the default communication interface and direction are set to `InteriorBorder_InteriorBorder_Interface` and `ForwardCommunication`
`LagrangeDiscreteFunctionSpace (`$\mathcal{G}$`,`if`,`dir`)`	creates a Lagrange discrete function space on $\mathcal{G}$; the default communication interface and direction are set to `if` and `dir`

Class 18 (`LagrangeDiscreteFunctionSpace`)

Class 19 (`DiscreteFunction` $\longrightarrow$ `GridFunction`)

The `DiscreteFunction` represents the interface for a discrete function $u_{\mathcal{G}} \in \mathcal{D}_{\mathcal{G}}$ (see Definition 19).

Template Parameters	
`Traits`	traits class from which the types are extracted

Exported Types	
`DiscreteFunctionSpaceType`	type of discrete function space $\mathcal{D}_{\mathcal{G}}$
`LocalFunctionType`	type of local discrete function u_E

Exported Types	
`DoFType`	type of the DoFs; this should equal the RangeFieldType from the discrete function space
`DofIteratorType`	type of iterator over over all DoFs of the discrete function

Interface Methods	
`string name()`	returns the name of the discrete function
`int size()`	returns the number of DoFs stored for the discrete function, i.e., $\lvert\mathcal{B}_\mathcal{G}\rvert$
`DofIteratorType dbegin/dend()`	returns an iterator pair over the interval $[begin, end)$ containing all DoFs on this discrete function
`LocalFunctionType localFunction(`E`)`	returns the local discrete function u_E for the (codimension 0) entity E
`ThisType& operator+=(`$v_\mathcal{G}$`)`	adds the function $v_\mathcal{G} \in \mathcal{D}_\mathcal{G}$ to $u_\mathcal{G}$
`ThisType& operator-=(`$v_\mathcal{G}$`)`	substracts the function $v_\mathcal{G} \in \mathcal{D}_\mathcal{G}$ from $u_\mathcal{G}$
`ThisType& operator*=(`α`)`	multiplies $u_\mathcal{G}$ by a scalar α
`ThisType& operator/=(`α`)`	divides $u_\mathcal{G}$ by a scalar α
`void addScaled(`$v_\mathcal{G}, \alpha$`)`	adds $\alpha \cdot v_\mathcal{G}$ to $u_\mathcal{G}$, where $v_\mathcal{G} \in \mathcal{D}_\mathcal{G}$ is a discrete function and α is a scalar
`void assign(`$v_\mathcal{G}$`)`	sets $u_\mathcal{G} = v_\mathcal{G}$
`void clear()`	sets $u_\mathcal{G} = 0$
`void enableDofCompression()`	enables compression of the internal DoF storage (automatically done when discrete function is managed by a `RestrictionProlongation` object during grid adaptation, see Class 39)

Class 19 (`DiscreteFunction`)

Class 20 (`LocalDiscreteFunction` $\longrightarrow$ `LocalFunction`)

The `LocalDiscreteFunction` represents the interface for a local function u_E of $u_\mathcal{G}$ (see Definition 20).

Template Parameters	
`Traits`	traits class from which the types are extracted

Exported Types and Constants	
	All types and constant from `FunctionSpace` are forwarded
`EntityType`	type of codimension 0 entity E
`BaseFunctionSetType`	type of base function set $\mathcal{B}_E$
`DiscreteFunctionSpaceType`	type of discrete function space $\mathcal{D}_\mathcal{G}$

Interface Methods	
`int numDofs()`	returns $\lvert I_E\rvert$, i.e., the number of local DoFs (see Definition 20)
`RangeFieldType& operator[](`i`)`	return a reference to local DoF u_i
`BaseFunctionSetType& baseFunctionSet()`	returns a reference to the base function set $\mathcal{B}_E$

Interface Methods	
`EntityType& entity()`	returns a reference to the entity E
`void evaluate(`$\hat{\lambda}$`,`val`)`	evaluates u_E in $\hat{\lambda} \in \hat{E}$, i.e., $val = u_E(F_E(\hat{\lambda}))$; for a localized base function set, this can be implemented by $val = \sum_i u_i \hat{\varphi}_i(\hat{\lambda})$
`void jacobian(`$\hat{\lambda}$`,`val`)`	evaluates the Jacobian of u_E, i.e., $val = \mathrm{D}u_E(F_E(\hat{\lambda}))$; for a localized base function set, this can be implemented by $val = \sum_i u_i \big(DF^{-T}(\hat{\lambda})\, \mathrm{D}\hat{\varphi}_i(\hat{\lambda})\big)$
`ThisType& operator+=(`v_E`)`	adds the local function v_E to u_E; this may affect local functions on neighboring elements
`ThisType& operator-=(`v_E`)`	substracts the local function v_E from u_E; this may affect local functions on neighboring elements
`void axpy(`α, v_E`)`	adds $\alpha \cdot v_E$ to $u_e lem$, where v_E is a local function and α is a scalar; this may affect local functions on neighboring elements
`void axpy(`$\hat{\lambda}, \boldsymbol{f}$`)`	adds $\boldsymbol{f} \cdot \varphi_i(\hat{\lambda})$ to each u_i; this may affect local functions on neighboring elements
`void axpy(`$\hat{\lambda}, \boldsymbol{j}$`)`	adds $\boldsymbol{j} \cdot \nabla\varphi_i(\hat{\lambda})$ to each u_i; this may affect local functions on neighboring elements
`void axpy(`$\hat{\lambda}, \boldsymbol{f}, \boldsymbol{j}$`)`	adds $\boldsymbol{f} \cdot \varphi_i(\hat{\lambda}) + \boldsymbol{j} \cdot \nabla\varphi_i(\hat{\lambda})$ to each u_i; this may affect local functions on neighboring elements
`void assign(`v_E`)`	sets $u_E = v_E$; this may affect local functions on neighboring elements
`void clear()`	sets $u_E = 0$; this may affect local functions on neighboring elements

Class 20 (`LocalDiscreteFunction`)

A.3 Discrete spatial operators

Class 21 (`Operator`)

The class `Operator` prescribes the interface for a general operator $L : V \longrightarrow W$ that maps from one function space V to another function space W.

Template Parameters	
`DomainField`	C++ type modeling the field of the domain V
`RangeField`	C++ type modeling the field of the range W
`Domain`	type of an element of the domain V
`Range`	type of an element of the range W

Interface Methods	
`virtual void operator()(`v, w`)`	apply the operator to $v \in V$ and store result in $w \in W$, i.e., $L(v) = w$

Class 21 (`Operator`)

In DUNE-FEM currently the following projection operators, all fulfilling the `Operator` interface, are implemented

- `L2Projection` implementing the L^2 projection described in Definition 32.
- `HdivProjection` implementing the projection of a discrete DG velocity such that the velocity field is continuous into normal direction across element faces (see [7]).
- `LagrangeProjection` implementing the projection of (possibly discontinuous) data into an appropriate Lagrange space and thus making it continuous. For the special case of continuous data, this coincides with the Lagrange interpolation (see Definition 31).

Class 22 (`InverseLinearOperator` $\longrightarrow$ `Operator`)

Given a linear `Operator` L, the class `InverseLinearOperator` describes the interface for the inverse operator $L^{-1} : \mathcal{D}_\mathcal{G} \longrightarrow \mathcal{D}_\mathcal{G}$ (see Definition 33).

Template Parameters	
`DiscreteFunction`	type of element $u_\mathcal{G} \in \mathcal{D}_\mathcal{G}$
`Operator`	type of linear operator L

Constructors	
`InverseLinearOperator` `(`L`,`r`,`ϵ`,`m`,`*verbose*`)`	create an operator inverting L; r is an error reduction that should be achieved, ϵ is a limit on the total error that should be achieved, m is the maximum number of iterations that should be done, and *verbose* is a flag for verbosity of the solver

Interface Methods	
`virtual void` `operator()(`$v_\mathcal{G}, u_\mathcal{G}$`)`	solve the system $L(u_\mathcal{G}) = v_\mathcal{G}$

Class 22 (`InverseLinearOperator`)

Class 23 (`OEMMatrix`)

Any matrix that shall be inverted using the OEM solvers has to satisfy the interface `OEMMatrix`.

Interface Methods	
`void` `multOEM(`v`,`w`)`	multiplies the matrix by v and stores result in w; both, v and w are of type `double*`
`double` `ddotOEM(`v`,`w`)`	evaluates the Euclidian scalar product $v \cdot w$ between the two vectors v ad w, which are of type `double*`
`void` `precondition(`v`,`w`)`	applies the preconditioning method to v and stores the result in w; both, v and w are of type `double*`

Class 23 (`OEMMatrix`)

Class 24 (`ISTLMatrix`)

Any matrix that shall be inverted using the solvers from DUNE-ISTL has to satisfy the interface `ISTLMatrix`.

Exported Types	
`MatrixAdapterType`	type of matrix adapter fulfilling the `MatrixAdapter` interface from DUNE-ISTL.

Interface Methods	
`MatrixAdapterType` `matrixAdapter()`	returns a matrix adapter that can be plugged into a linear solver provided by DUNE-ISTL. The provided implementation of DUNE-FEM also stores the preconditioning fulfilling the `Preconditioner` interface from DUNE-ISTL and a proper scalar product of type `ScalarProduct` from DUNE-ISTL.

Class 24 (`ISTLMatrix`)

Class 25 (`LocalLinearOperator`)

The class `LocalLinearOperator` describes the interface for local matrices (see Definition 35).

Template Parameters	
`Traits`	traits structure holding all necessary type information

Interface Methods	
`void init(`E, E_e`)`	initialize the local linear operator for a pair of entities, for example, element E and neighbor E_e over intersection e
`int rows()`	returns $\|I_E\|$, the number of rows of the local linear operator which is the same as the number of base functions from `DomainBaseFunctionSet`
`int columns()`	returns $\|I_{E_e}\|$, the number of columns of the local linear operator which is the same as the number of base functions from `RangeBaseFunctionSet`
`void clear()`	set all entries to zero
`void resort()`	if the matrix implementation provides this feature, then resort appearance of entries in ascending order
`void add(`r`,`c`,`v`)`	add v to entry (r, c), $r \in \{0, \dots, \|I_E\| - 1\}$, $c \in \{0, \dots, \|I_{E_e}\| - 1\}$
`void set(`r`,`c`,`v`)`	set entry (r, c) to value v, $r \in \{0, \dots, \|I_E\| - 1\}$, $c \in \{0, \dots, \|I_{E_e}\| - 1\}$
`RangeFieldType get(`r`,`c`)`	return value of entry (r, c), $r \in \{0, \dots, \|I_E\| - 1\}$, $c \in \{0, \dots, \|I_{E_e}\| - 1\}$
`void scale(`v`)`	scale all entries of the local linear operator with value v
`void multiplyAdd(`f_d`, `f_r`)`	evaluate $f_r = L \cdot f_d$ where f_d is a local function of a discrete function from `DomainSpace` and f_r a local function of a discrete function from `RangeSpace`
`void unitRow(`r`)`	set all entries of row r to zero except the diagonal entry which is set to 1, $r \in \{0, \dots, \|I_E\| - 1\}$
`DomainSpace& domainSpace()`	return reference to `DomainSpace`
`RangeSpace& rangeSpace()`	return reference to `RangeSpace`
`DomainBaseFunctionSet& domainBaseFunctionSet()`	return reference to the base function set for `DomainEntity`
`RangeBaseFunctionSet& rangeBaseFunctionSet()`	return reference to the base function set for `RangeEntity`

Class 25 (`LocalLinearOperator`)

Class 26 (`LinearOperator` $\longrightarrow$ `Operator`)

The class `LinearOperator` describes the interface for linear discrete operators in DUNE-FEM (see Definition 34). These linear discrete operators are representable by a matrix.

Template Parameters	
`DomainSpace`	type of discrete function space describing the domain of the operator
`RangeSpace`	type of discrete function space describing the range of the operator
`Traits`	traits structure holding information about the local stencil of the matrix

Interface Methods	
`LocalLinearOperator localLinearOperator(`E, N`)`	return local linear operator belonging to entity combination E, N, where global DoF numbers from E correspond to the rows of the matrix and the global DoF numbers from N to the column numbers.
`void clear()`	set all entries of internal matrix to zero

Interface Methods	
`void reserve()`	reserve memory for internal matrix

Class 26 (`LinearOperator`)

Class 27 (`LinearOperatorImplementation` ⟶ `LinearOperator`)

Here we present the constructor for a linear operator which is the same for all implementations in DUNE-FEM (see also Definition 34).

Constructors	
`LinearOperatorImplementation` ($\mathcal{D}_\mathcal{G}^V$, $\mathcal{D}_\mathcal{G}^W$)	common constructor for linear operators, the domain discrete function space $\mathcal{D}_\mathcal{G}^V$ describes the unknowns for the rows and the range discrete function space $\mathcal{D}_\mathcal{G}^W$ the unknowns corresponding the the columns

Class 27 (`LinearOperatorImplementation`)

Class 28 (`Pass` ⟶ `Operator`)

The class `Pass` provides an interface for passes of combined operators (see Definition 37).

Template Parameters	
`Traits`	traits class containing the types of the range and destination discrete function spaces
`PreviousPass`	type of the previous pass in the pass list
`passId`	integer ID of this pass

Constructors	
`Pass(previousPass)`	implementation of the previous pass

Interface Methods	
`void operator()` ($v_\mathcal{G}, u_\mathcal{G}$)	evaluate all previous passes in the list and use the result for the computation of this pass

Class 28 (`Pass`)

A.4 Parallelization and data exchange

Class 29 (`CommunicationManager`)

The `CommunicationManager` is in charge of doing communications in parallel simulations. We group all communications for discrete functions belonging to the same discrete function space since the communication pattern is in most cases the same. The user still has the flexibility to exchange the data in a different way if needed. Each discrete function space defines a default communication pattern and an operation to be formed on the data.

Template Parameters	
`DiscreteFunctionSpace`	discrete function space $\mathcal{D}_\mathcal{G}$ that describes discrete function the communication is done for

Interface Methods	
`void exchange(`$u_\mathcal{G}$`)`	exchange data of discrete function $u_\mathcal{G}$ according to the space's default communication behavior and default operation type

Interface Methods	
`void exchange(`$u_{\mathcal{G}}$`,`γ`)`	exchange data of discrete function $u_{\mathcal{G}}$ according to the space's default communication behavior and an operation γ. For example γ could be a `+=` (summation) or `=` (copying) or other operations.

Class 29 (`CommunicationManager`)

A.5 Adaptation and load-balancing

Class 30 (`ManagedIndexSet`)

The `DofManager` needs to know all persistent and consecutive persistent index sets depending on the grid instance $\mathcal{H}$ the `DofManager` is responsible for. Therefore, a list of `ManagedIndexSet` are stored. These are simple wrapper classes storing references to the real index sets. The interface methods are realized via virtual methods. A `ManagedIndexSet` is created by the call to the method `addIndexSet` of the `DofManager`.

Interface Methods	
`virtual void resize()`	resizes the index set to current state of the grid
`virtual bool compress()`	compresses the index set; returns `true` if the index set was not in compressed state
`void read(filename,`n`)`	reads back the index set for time step n from a file named `filename`
`void write(filename,`n`)`	writes the index set for time step n to a file named `filename`
`bool increaseReference(`$\Lambda_{\mathcal{G}}$`)`	returns `true` if $\Lambda_{\mathcal{G}}$ coincides with this index set and increases the internal reference counter accordingly
`bool decreaseReference(`$\Lambda_{\mathcal{G}}$`)`	return `true` if $\Lambda_{\mathcal{G}}$ coincides with this index set and no other references exist

Class 30 (`ManagedIndexSet`)

Class 31 (`DofStorageInterface`)

The `DofStorageInterface` is the interface for an unmanged DoF storage.

Interface Methods	
`virtual std::string name()`	returns the name of DoF storage, i.e., the name of the discrete function
`virtual void enableDofCompression()`	enables DoF compression for this DoF storage, see `DiscreteFunction` (Class 19)

Class 31 (`DofStorageInterface`)

Class 32 (`ManagedDofStorageInterface` $\longrightarrow$ `DofStorageInterface`)

The `ManagedDofStorageInterface` inherits the `DofStorageInterface` and provides an interface for a DoF storage that should be managed by the `DofManager`. This means during the adaptation process the memory of these DoF storages is resized and compressed if necessary (see also Definition 57).

Interface Methods	
`virtual int size()`	return current size of DoF array
`virtual void resize()`	resize the memory according to the provided size from the corresponding DoF mapper

Interface Methods	
`virtual void reserve(`s`)`	reserve memory for at least s entries
`virtual void dofCompress()`	Compress DoF array by moving entries from rear positions to unused positions ahead. This is only executed if the underlying index set is consecutive (see Definition 12, 57 and Example 58).
`virtual void enableDofCompression()`	enables DoF compression for this DoF storage. This is for example done for a DoF storage that is passed to a `RestrictProlongDefault` object or marked for redistribution during load balancing (see Class 36).
`virtual size_t usedMemorySize()`	returns number of bytes of the memory occupied by this memory object

Class 32 (`ManagedDofStorageInterface`)

Class 33 (`ManagedDofStorage` $\longrightarrow$ `ManagedDofStorageInterface`)

The `ManagedDofStorage` inherits the `ManagedDofStorageInterface` and provides an implementation for the creation of a DoF storage that is managed by the `DofManager`.

Template Parameters	
`Grid`	type of the hierarchal grid $\mathcal{H}$ (for access to the `DofManager`)
`Mapper`	type of the DoF mapper for this storage
`DofArray`	type of DoF container, e.g., `double*` or `std::vector<double>`

Constructors	
`ManagedDofStorage(`$\mathcal{H}$`,`$\mu_{\mathcal{G}}$`,name)`	creates a DoF container of type `DofArray` for the number of unknowns obtained from $\mu_{\mathcal{G}}$ and makes it known to the `DofManager` by calling its `addDofStorage` method; the container is automatically removed from the `DofManager` on destruction

Additional Interface Methods	
`DofArray& getArray()`	returns a reference to the DoF storage
`void moveToRear()`	move memory blocks to rear position to make insertion possible
`void moveToFront()`	move memory blocks to front position to return to compressed state

Class 33 (`ManagedDofStorage`)

Class 34 (`DofManager`)

The `DofManager` is an implementation for the central management of memory needed for the storage of user data, i.e., the DoFs and index sets created. To ensure that only one instance of a `DofManager` for a hierarchic grid $\mathcal{H}$ exists, the `DofManager` can only be accessed via a static method [none]instance($\mathcal{H}$) which implements the **singleton per unique key** concept. Given a hierarchic grid $\mathcal{H}$, the method `instance` returns a reference to the associated `DofManager`.

Template Parameters	
`Grid`	type of the hierarchic grid $\mathcal{H}$

Exported Types	
`GridType`	type of the hierarchic grid $\mathcal{H}$

Interface Methods	
`void addIndexSet(`$\Lambda_\mathcal{G}$`)`	makes a persistent index set $\Lambda_\mathcal{G}$ known to the `DofManager`; this method is automatically called on construction of a pesistent index set
`void removeIndexSet(`$\Lambda_\mathcal{G}$`)`	removes a persistent index set $\Lambda_\mathcal{G}$ from `DofManager`'s internal list; this method is automatically called on destruction of a persistent index set
`void addDofStorage(`ds`)`	adds a DoF storage ds to the `DofManager`'s list of managed DoF storages
`void removeDofStorage(`ds`)`	removes a DoF storage ds from `DofManager`'s list of managed DoF storages
`void resize()`	triggers the resize process; first all index sets are resized, then all managed DoF storages
`void compress()`	triggers the compression process; first for all index sets are compressed, then all managed DoF storages
`void writeIndexSets(filename,`n`)`	writes all managed index sets for time step n to a file named `filename`
`void readIndexSets(filename,`n`)`	reads back all index sets for time step n from a file named `filename`
`int sequence()`	for a hierarchic grid $\mathcal{H}^n$, the sequence number n is returned; this feature will directly be supported by future versions of the DUNE grid interface

Class 34 (`DofManager`)

Class 35 (`LoadBalancerInterface`)

The `LoadBalancerInterface` represents the interface for the load-balancing process described in Definition 40.

Interface Methods	
`void loadBalance()`	triggers the recalculation of the master decomposition

Class 35 (`LoadBalancerInterface`)

Class 36 (`LoadBalancer` $\longrightarrow$ `LoadBalancerInterface`)

The class `LoadBalancer` implements the `LoadBalancerInterface`.

Template Parameters	
`Grid`	type of hierarchical grid $\mathcal{H}$ to be dynamically load balanced

Constructors	
`LoadBalancer(`$\mathcal{H}$`)`	creates a `LoadBalancer` for the hierarchical grid $\mathcal{H}$
`LoadBalancer(`$\mathcal{H}$`,`rp`)`	creates a `LoadBalancer` for the hierarchical grid $\mathcal{H}$ and adds all functions given by the restriction/prolongation operator rp

Additional Interface Methods	
`void addDiscreteFunction(`$u_\mathcal{G}$`)`	add $u_\mathcal{G}$ to list of discrete functions that are redistributed together with the recalculatiuon of the master decomposition. DoF compression is enabled for $u_\mathcal{G}$.

Class 36 (`LoadBalancer`)

Class 37 (`AdaptationManagerInterface` $\longrightarrow$ `LoadBalancerInterface`)

The `AdaptationManagerInterface` represents the interface for the adaptation process described in Definition 40.

Template Parameters	
`Grid`	type of hierarchical grid $\mathcal{H}$ for which adaptation is done
`RestProlOperator`	type of the restriction and prolongation operator that transfers the data from $\mathcal{H}^n$ to $\mathcal{H}^{n+1}$, this class has to satisfy the `RestrictionProlongation`.

Interface Methods	
`bool adaptive()`	returns `true` if adaptation is possible for the given grid $\mathcal{H}$ and also enabled
`void adapt()`	trigger the adaptation process which will convert $\mathcal{H}^n$ into $\mathcal{H}^{n+1}$ and also trigger the restriction and prolongation process. See Description 74 for details.

Class 37 (`AdaptationManagerInterface`)

Class 38 (`AdaptationManager` $\longrightarrow$ `SerialAdaptationManager, LoadBalancer`)

The `AdaptationManager` class inherits from the `SerialAdaptationManager` as well as from `LoadBalancer`. A reimplementation of the method `adapt` is done.

Template Parameters	
`Grid`	type of hierarchical grid $\mathcal{H}$ for which adaptation is done
`RestProlOperator`	type of the restriction and prolongation operator that transfers the data from $\mathcal{H}^n$ to $\mathcal{H}^{n+1}$; this class has to satisfy the `RestrictionProlongation` interface.

Reimplemented Methods	
`void adapt()`	calls first the `adapt` method from `SerialAdaptationManager` doing the grid adaptation including restriction and prolongation and second the `loadBalance` method from `LoadBalancer` to trigger the load balancing process

Class 38 (`AdaptationManager`)

Class 39 (`RestrictionProlongation`)

The `RestrictionProlongation` represents the interface for a local restriction and prolongation operator.

Template Parameters	
`DiscreteFunction`	type of DiscreteFunction $u_\mathcal{G}$ the restriction and prolongation is done for

Interface Methods	
`void restrictLocal(`E_f`, `E`, flag)`	apply local restriction operator for E_f (see Definition 42), if `flag` is `true` then initialization of u_{E_f} is done
`void prolongLocal(`E`, `E_s`, flag)`	apply local prolongation operator for E_s (see Definition 44), if `flag` is `true` then initialization of u_{E_s} is done
`void setFatherChildWeight(`ω`)`	set weight of $\lvert e\rvert/\lvert E\rvert$ where $e \in \mathcal{C}^1_E(E)$ and the hierarchy grid contains only one geometry type, for example only simplicial elements, otherwise the weight is calculated new for each pair (e, E)

Class 39 (`RestrictionProlongation`)

Class 40 (`RestrictProlongDefault` $\longrightarrow$ `RestrictionProlongaion`)

The `RestrictProlongDefault` class implements the `RestrictionProlongation` interface. For each discrete function space implementation there is one such default implementation of the `RestrictionProlongation` interface.

Template Parameters	
`DiscreteFunction`	type of DiscreteFunction $u_{\mathcal{G}}$ the restriction and prolongation is done for

Constructors	
`RestrictProlongDefault(` $u_{\mathcal{G}}$ `)`	Create restriction/prolongation operator for the discrete function $u_{\mathcal{G}}$. DoF compression for $u_{\mathcal{G}}$ in enabled hereby.

Class 40 (`RestrictProlongDefault`)

A.6 Time discretization

Class 41 (`TimeProvider`)

The class `TimeProvider` defines the interface for classes providing a simulation time, a time step size, CFL number and so on, for the use in non-stationary simulations.

Template Parameters	
`CollectiveCommunication`	type of collective communication for synchronization of global information, e.g., time step size

Interface Methods	
`int timeStep()`	returns n
`double time()`	returns t^n
`double deltaT()`	returns Δt^n
`void provideTimeStepEstimate(`$\Delta t_{\mathcal{L}_{\mathcal{G}}}$`)`	provides a time step estimate $\Delta t_{\mathcal{L}_{\mathcal{G}}}$ given by stabilization criteria from the space discretization operator $\mathcal{L}_{\mathcal{G}}$
`void init()`	initializes time provider with a large number as time step size
`void init(`Δt`)`	initializes time provider with given Δt
`void next()`	advances to next time step; this involves a global communication to compute the minimal global time step size
`double factor()`	returns c_{cfl}

Class 41 (`TimeProvider`)

Class 42 (`SpaceOperator` $\longrightarrow$ `Operator`)

The class `SpaceOperator` inherits the `Operator` class, specifying that $V = W$, i.e., $\mathcal{L}_{\mathcal{G}} : V \longrightarrow V$ for some discrete function space V. This operator also represents the interface for spatial discretization operators used with ODE solvers.

Template Parameters	
`DiscreteFunction`	type of discrete function that this operator is applied to, i.e., $u_{\mathcal{G}} \in \mathcal{D}_{\mathcal{G}}$

Exported Types	
`SpaceType`	type of discrete function space $\mathcal{D}_\mathcal{G}$ extracted from `DiscreteFunction`

Interface Methods	
`virtual void operator()`$(v_\mathcal{G}, u_\mathcal{G})$	applies the operator to $v_\mathcal{G} \in \mathcal{D}_\mathcal{G}$ and stores the result in $u_\mathcal{G} \in \mathcal{D}_\mathcal{G}$, i.e., $\mathcal{L}_\mathcal{G}(v_\mathcal{G}) = u_\mathcal{G}$
`SpaceType& space()`	return reference to discrete function space $\mathcal{D}_\mathcal{G}$
`void setTime(`t`)`	set current simulation time t to operator $\mathcal{L}_\mathcal{G}$
`double timeStepEstimate()`	return maximum possible time step size Δt that ensures a stable explicit Euler solver

Class 42 (`SpaceOperator`)

Class 43 (`OdeSolver`)

The class `OdeSolver` describes the interface for an ODE solver.

Template Parameters	
`Destination`	type of discrete function that this ODE solver is applied to

Interface Methods	
`void initialize(`$u_\mathcal{G}$`)`	initialize ODE solver for example to determine an initial time step estimate
`void solve(`$u_\mathcal{G}$`)`	solve $\partial_t u_\mathcal{G} = \mathcal{L}_\mathcal{G}(u_\mathcal{G})$. $u_\mathcal{G}$ is overwritten with the result of the solution process.

Class 43 (`OdeSolver`)

Class 44 (`ODE solvers` $\longrightarrow$ `OdeSolver`)

The ODE solvers `ExplicitRungeKuttaSolver`, `ExplicitOdeSolver`, or `ImplicitOdeSolver` have exactly the same constructor parameter list. For the `SemiImplicityOdeSolver` two instead of one spatial discretization operator has to be provided.

Template Parameters	
`Destination`	type of discrete function that this ODE solver is applied to

Constructors for `ExplicitRungeKuttaSolver`, `ExplicitOdeSolver`, and `ImplicitOdeSolver`	
`ODESolver(`$\mathcal{L}_\mathcal{G}$`, tp, `k`, flag)`	Create an ODE solver. $\mathcal{L}_\mathcal{G}$ is the spatial discretization operator, `tp` is the `TimeProvider` managing simulation time and time step size, k describes the desired order of the solver, and `flag` is a verbosity flag which defaults to `false`

Constructors for `SemiImplicitOdeSolver`	
`SemiImplicityOdeSolver (`$\mathcal{L}_\text{expl}$`, `$\mathcal{L}_\text{impl}$`, tp, `k`, flag)`	Create an `SemiImplicityOdeSolver`. $\mathcal{L}_\text{expl}$ is the explicit spatial discretization operator and $\mathcal{L}_\text{impl}$ represents the implicit spatial discretization operator. `tp` is the `TimeProvider` managing simulation time and time step size, k describes the desired order of the solver, and `flag` is a verbosity flag which defaults to `false`

Class 44 (`ODE solvers`)

References

[1] A. Alexandrescu. *Modern C++ Design.* Addison Wesley, 2001.

[2] R.E. Bank, A.H. Sherman, and Weiser A. Some refinement algorithms and data structures for regular local mesh refinement. In *Scientific Computing. The mathematics of finite elements and applications. Highlights 1993. Proceedings of the 8th conference on the mathematics of finite elements and applications, MAFELAP '93, held at Brunel University, Uxbridge, UK, April 27-30, 1993*, pages 3–17. IMACS North Holland, 1983.

[3] E. Bänsch. Local mesh refinement in 2 and 3 dimensions. *IMPACT Comput. Sci. Eng.*, 3(3):181–191, 1991.

[4] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, and C. Wieners. UG - a flexible software toolbox for solving partial differential equations. *Computing and Visualization in Science*, 1:27–40, 1997.

[5] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. II: Implementation and tests in dune. *Computing*, 82(2-3):121–138, 2008.

[6] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. I: Abstract framework. *Computing*, 82(2-3):103–119, 2008.

[7] P. Bastian and B. Rivière. Superconvergence and $H(div)$ projection for discontinuous Galerkin methods. *Int. J. Numer. Methods Fluids*, 42(10):1043–1057, 2003.

[8] J. Bey. Tetrahedral grid refinement. *Computing*, 55(4):355–378, 1995.

[9] J. Bey. Simplicial grid refinement: On Freudenthal's algorithm and the optimal number of congruence classes. *Numer. Math.*, 85(1):1–29, 2000.

[10] M. Blatt and P. Bastian. The iterative solver template library. In B. Kagström, E. Elmroth, J. Dongarra, and J. Wasniewski, editors, *Applied Parallel Computing – State of the Art in Scientific Computing*, pages 666–675, Berlin/Heidelberg, 2007. Springer.

[11] A. Burri, A. Dedner, D. Diehl, R. Klöfkorn, and M. Ohlberger. A general object oriented framework for discretizing nonlinear evolution equations. In Y.I. Shokin, M. Resch, N. Danaev, M. Orunkhanov, and N. Shokina, editors, *Advances in High Performance Computing and Computational Sciences*, Berlin/Heidelberg, 2006. Springer.

[12] B. Cockburn, G. Kanschat, and D. Schötzau. A locally conservative ldg method for the incompressible navier-stokes equations. *Math. Comput.*, 74(251):1067–1095, 2005.

[13] B. Cockburn and C.-W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws V: Multidimensional systems. *J. Comput. Phys.*, 141:199–224, 1998.

[14] F. Darema. The SPMD model: Past, present and future. Cotronis, Yiannis (ed.) et al., Recent advances in parallel virtual machine and message passing interface. 8th European PVM/MPI user's group meeting Santorini/ Thera, Greece, September 23-26, 2001. Proceedings. Berlin: Springer. Lect. Notes Comput. Sci. 2131, 1, 2001.

[15] T. A. Davis. Algorithm 832: UMFPACK v4.3—an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw.*, 30(2):196–199, 2004.

[16] A. Dedner and R. Klöfkorn. A generic stabilization approach for higher order Discontinuous Galerkin methods for convection dominated problems. Preprint no. 8 (submitted to SIAM Sci. Comput.), Mathematisches Institut, Unversität Freiburg, 2008. http://www.mathematik.uni-freiburg.de/IAM/homepages/robertk/postscript/dedner_kloefkorn_limiter.pdf.

[17] A. Dedner, M. Luethi, T. Albrecht, and T. Vetter. Curvature Guided Level Set Registration using Adaptive Finite Elements. In F. Hamprecht, C. Schnorr, and B. Jahne, editors, *Proc. of the 29th Annual Symposium of the German Association for Pattern Recognition*, Berlin/Heidelberg, 2007. Springer.

[18] A. Dedner, C. Rohde, B. Schupp, and M. Wesenberg. A parallel, load balanced mhd code on locally adapted, unstructured grids in 3d. *Computing and Visualization in Science*, 7:79–96, 2004.

[19] D. Diehl. *Higher order schemes for simulation of compressible liquid-vapor flows with phase change*. PhD thesis, Universität Freiburg, 2007. http://www.freidok.uni-freiburg.de/volltexte/3762/.

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

[21] J.-F. Gerbeau and B. Perthame. Derivation of viscous Saint-Venant system for laminar shallow water; numerical validation. *Discrete Contin. Dyn. Syst., Ser. B*, 1(1):89–102, 2001.

[22] C. Gersbacher. Local discontiunous galerkin verfahren zur simulation flacher dreidimensionaler strömungen mit freier oberfläche. Master's thesis, Universität Freiburg, 2008.

[23] T. Gessner, B. Haasdonk, R. Kende, M. Lenz, M. Metscher, R. Neubauer, M. Ohlberger, W. Rosenbaum, M. Rumpf, R. Schwörer, R. Spielberg, and U. Weikard. A procedural interface for multiresolutional visualization of general numerical data. Technical Report 28, SFB 256, Bonn, 1999. http://www.math.uni-muenster.de/u/ohlberger/postscript/hmesh.ps.gz.

[24] S. Gottlieb, C.-W. Shu, and E. Tadmor. Strong stability-preserving high-order time discretization methods. *SIAM Rev.*, 43(1):89–112, 2001.

[25] M. Griebel and G. Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Comput.*, 25(7):827–843, 1999.

[26] N. M. Josuttis. *The C++ Standard Library*. Addison-Wesley Professional, 1999.

[27] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *SIAM Rev.*, 41(2):278–300, 1999.

[28] M. Kränkel. Local Discontinuous Galerkin Methoden für elliptische Differentialgleichungen und das Stokes System. Master's thesis, Universität Freiburg, 2008.

[29] D. Kröner. *Numerical Schemes for Conservation Laws*. Verlag Wiley & Teubner, Stuttgart, 1997.

[30] M.L. Lehn. *FLENS A flexible library for efficient numerical solutions*. PhD thesis, Fakultät für Mathematik und Wirtschaftswissenschaften, Universität Ulm, 2008. http://flens.sourceforge.net/.

[31] R. J. Leveque. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge: Cambridge University Press., 2002.

[32] W. F. Mitchell. A comparison of adaptive refinement techniques for elliptic problems. *ACM Trans. Math. Softw.*, 15(4):326–347, 1989.

[33] A. Schmidt and K.G. Siebert. *Design of Adaptive Finite Element Software – The Finite Element Toolbox ALBERTA*. Springer, 2005.

[34] B. Schupp. *Entwicklung eines effizienten Verfahrens zur Simulation kompressibler Strömungen in 3D auf Parallelrechnern*. PhD thesis, Universität Freiburg, 1999. http://www.freidok.uni-freiburg.de/volltexte/68/.

[35] Website. DUNE-FEM – The FEM Module. http://dune.mathematik.uni-freiburg.de/.

[36] Website. ALUGrid – Adaptive, Load-balanced, and Unstructured Grid library. http://www.mathematik.uni-freiburg.de/IAM/Research/alugrid/.

[37] Website. Gnuplot. http://www.gnuplot.info/.

[38] Website. METIS. http://glaros.dtc.umn.edu/gkhome/views/metis/.

[39] Website. ParaView. http://www.paraview.org/.

[40] Website. Scientific computing in object-oriented languages. http://www.oonumerics.org/oon/.

[41] Website. UMFPACK. http://www.cise.ufl.edu/research/sparse/umfpack/.

[42] Website. VISIT. https://wci.llnl.gov/codes/visit/.

[43] Website. VTK. http://www.vtk.org/.