A hash data structure for adaptive PDE–solvers based on Discontinuous Galerkin discretizations

Kolja Brix, Ralf Massjung und Alexander Voss

Bericht Nr. 302 **August 2009**

Key words: Discontinuous Galerkin method, adaptive method, multilevel method, pointerless data structures, neighboring algorithm

AMS subject classifications: 65N30, 65N50, 68N01, 68P05

Institut für Geometrie und Praktische Mathematik
RWTH Aachen

Templergraben 55, D–52056 Aachen (Germany)

A HASH DATA STRUCTURE FOR ADAPTIVE PDE–SOLVERS BASED ON DISCONTINUOUS GALERKIN DISCRETIZATIONS

KOLJA BRIX*, RALF MASSJUNG*, AND ALEXANDER VOSS*

Abstract. Adaptive multiscale methods are among the most effective techniques for the numerical solution of partial differential equations. Efficient grid management is an important task in these solvers. In this paper we focus on this problem for Discontinuous Galerkin discretization methods in 2 and 3 spatial dimensions and present a data structure for handling adaptive grids of different cell types in a unified approach. Instead of tree-based techniques where connectivity is stored via pointers, we associate each cell that arises in the refinement hierarchy with a cell identifier, and construct algorithms that establish hierarchical and spatial connectivity. By means of bitwise operations, the complexity of the connectivity algorithms can be bounded independent of the level. The grid is represented by a hash table which results in a low-memory data structure and ensures fast access to cell data. The spatial connectivity algorithm also supports the application of quadrature rules for face integrals that occur in Discontinuous Galerkin discretizations. The concept allows to implement Discontinuous Galerkin methods largely independent of spatial dimension and cell type. We demonstrate this by outlining how typical algorithmic tasks that arise in these implementations can be performed with our data structure.

Key words. Discontinuous Galerkin method, adaptive method, multilevel method, pointerless data structures, neighboring algorithm

AMS subject classifications. 65N30, 65N50, 68N01, 68P05

1. Introduction. In order to approximate solutions of partial differential equations (PDE) efficiently, the most advanced numerical algorithms rely on adaptive as well as multilevel techniques [13]. To further accelerate computations, these techniques are typically combined with a parallelization [3, 30]. Adaptive multilevel methods require operations on the locally adapted discretization grid and on the grid hierarchy underlying the multilevel structure. To support these techniques, it is important to have a low-memory data structure to store the adaptive grid hierarchy, a method to quickly change the grid due to refinement or coarsening needs and fast access mechanisms to provide the data stored on the grid cells and to determine the *hierarchical connectivity* (connectivity of parent and child cells) as well as the *spatial connectivity* (connectivity of spatially neighboring grid items). These requirements have been central to our development of a hash-based data structure for adaptive multilevel grids, which we present in this article. Here we have focused on creating a data structure which supports PDE-solvers that employ the Discontinuous Galerkin (DG) Method as discretization tool [2, 11, 1]. Additional aspects that we have incorporated are the flexibility of the data structure for use in conjunction with different cell types (2D and 3D grids; hybrid grids consisting of several cell types, e.g. triangles and quadrilaterals) and the support for evaluating quadrature formulas, which occur when assembling the discrete DG equations. An important consequence of concentrating on DG methods is that the only connectivity needed in assembling discrete equations is the connectivity of cells across faces. It is meanwhile well known, that DG methods can be used to successfully handle all common types of partial differential equations, are well suited for parallelization, have the convenient property of allowing hanging nodes and allow straightforwardly to vary the approximation order within the grid (p-adaptivity), see [11].

In contrast to the classical approach that employs pointer structures to store grid connectivities, a concept based on hash tables possesses the following features, which we consider as advantageous: It has lower storage requirements and does not need computational work to update connectivities while refining or coarsening the grid. The hash-based concept needs to calculate spatial connectivity, which we analyze here in a unified way independent of the cell type. We develop fast algorithms for calculating the spatial connectivity. Alltogether, this results in a grid management concept, which is implemented largely independent of spatial dimension and cell type. We summarize the core ingredients of our approach, which was initiated in [29], and relate it to the literature:

*Institut für Geometrie und Praktische Mathematik, RWTH Aachen, 52056 Aachen, Germany, {brix,massjung,voss}@igpm.rwth-aachen.de, http://www.igpm.rwth-aachen.de/{brix,massjung,voss}

Cell identifier and algorithm for spatial connectivity. The grid hierarchy is generated by a *refinement rule* that divides each cell of a given cell type into its child cells. Thus, we uniquely characterize each grid cell that may possibly occur within the grid hierarchy by a *cell identifier*. The cell identifier directly provides the hierarchical connectivity. The spatial connectivity, i.e. to find for a given cell identifier the identifiers of the corresponding neighboring cells, is examined theoretically in this article. The theory is presented independent of a particular cell type and results in an algorithm that delivers the spatial connectivity, provided that several properties are satisfied by the refinement rule. The validity of these properties are easily checked for standard refinement rules that we employ for triangles, tetrahedra, quadrilaterals, and cuboids. This unified algorithm turns out to have a rather simple structure and also works for hybrid grids and for cell types with refinement rules that are generated in a tensor-product fashion from other cell types, e.g. prisms are treated as tensor products of quadrilaterals and triangles. We apply this concept to solve PDEs with DG methods in the practically important case of graded adaptive grids [6, 7, 25].

The usage of cell identifiers for finding neighbors in recursively refined meshes has been studied in several previous works. In [14] a method is proposed to handle spherical data in geographic information systems. The surface of a sphere is approximated by an inscribed icosahedron, its surface representing a triangular net. Adaptively refined grids of this specific coarse grid are considered and the spatial connectivity is derived in terms of cell identifiers. In [24], the method of [14] is extended to the surface triangulations of the tetrahedron and the octahedron. In [23] the spatial connectivity is derived in terms of cell identifiers for adaptively refined tetrahedral grids. Only the case of a coarse grid consisting of six tetrahedra, which are generated by dividing a cuboid, is considered, while we do not have any restrictions on the coarse grid. The refinement rule used in [23] divides a tetrahedron into two subtetrahedra, whereas in our refinement rule, a tetrahedron is divided into eight subtetrahedra.

Stability of refinement rule for cell type tetrahedron. An indispensable property of the grids when utilized for DG methods, or other Finite Element methods, is that the grid cells do not exhibit arbitrarily small angles in the refinement process. This issue is particularly involved for tetrahedral grids and has been analyzed in [4], where the refinement process is proven to generate only a finite number of congruence classes of tetrahedra. The refinement rule proposed in [4] does not fulfill all the properties that we demand from a refinement rule. Our refinement rule employs a different numbering than the one of [4] and we have checked computationally that it generates also a finite number of congruence classes of tetrahedra.

Face quadrature. Assembling the discrete DG equations on a grid, integrals along the faces of the grid have to be evaluated. In order to reduce function evaluations, one typically stores the shape function values at the face quadrature points. Within our algorithm for the spatial connectivity, a single look up in a table provides the numberings of the face quadrature points with respect to both cells adjacent to the face. Again this is theoretically described in a unified manner for all cell types and also works for hybrid grids and graded adaptive grids. To our knowledge this point as not yet been addressed in the literature.

Hash table. We employ a hash table to store the *cell data*, which in the case of a DG method are the coefficients of the numerical solution associated with the cell. For all cell types, we convert the cell identifier into an unsigned integer, which serves as the key in the hash table. This results in a grid management independent of the cell type. Our hash table is based on chaining, which requires pointers to set up the chains. Furthermore, pointers are used to establish doubly-linked lists that connect entries of the hash table associated with the same level in the grid hierarchy. The storage requirements for the unsigned integer keys and the pointers within the hash table are low compared to classical concepts, which are based on tree structures providing the spatial as well as the hierarchical connectivity by pointers. Moreover, when refining or coarsening the grid, additional work to update the hierarchical and the spatial connectivity is needed in the tree based data structure concept.

A particular attempt to use hash data structures to create efficient parallel adaptive multilevel solvers was made in [16]. This work, which is restricted to Cartesian grids, points out that pointer based data structures are inferior to hash-based data structures with respect to storage requirements and computing time.

Bitwise operations. In the worst case, the complexity of the algorithm, that provides the spatial connectivity, is proportional to the level of the corresponding grid cells. We have removed this level dependence by replacing the corresponding steps in the algorithm by bitwise operators, which depend on the cell type, and act on the unsigned integer associated with the cell identifier. This is the same unsigned integer that is used as the key in the hash table. Employing the bitwise operators ensures that independent of the cells involved, the spatial connectivity is provided by a fixed, small number of bitwise operations, where a bitwise operation is typically applied within one processor clock cycle. This provides a fast access to cell neighbors.

Amongst the neighboring algorithms based on cell identifiers, that we have already mentioned above, [24] and [23] also express their algorithms in terms of bitwise operations in order to obtain algorithms with complexity independent of the level of the cells involved.

The paper is organized as follows: In Sect. 2 we first specify what kind of cell type we consider and how a corresponding refinement rule and a cell identifier is defined. We then turn to the problem of determining the spatial connectivity within the refinement hierarchy of a single cell. For this purpose we compile in Sec. 2.3 a number of properties to be demanded from cell type and refinement rule so that the spatial connectivity can be determined by the algorithm given in Sec. 2.4. Here we illustrate these concepts for straightforward refinement rules for the cell types triangle and tetrahedron. Sect. 2 is completed by discussing the stability of the refinement rule that we propose for tetrahedra and by supplementing the spatial connectivity algorithm with a strategy that supports face quadratures required to implement DG methods. Sect. 3 generalizes the above concepts to the refinement hierarchy of a coarse grid, where the coarse grid may be composed of grids of different cell types, and discusses simply graded grids. In Sect. 4 the hash data structure for storing grids and accessing grid data is discussed. Here we also show how the cell identifier is encoded as an unsigned integer and how bitwise operations on this unsigned integer are devised in order to turn the algorithm for the spatial connectivity into a form that has level-independent complexity. The section closes with a presentation of how to apply the data structure to perform several algorithmic tasks that occur in the implementation of DG methods. Finally in Appendix A we present the refinement rules for further cell types, namely for quadrilaterals, cuboids and prisms.

2. Refinement hierarchy and connectivity.

In this section we consider the refinement of a single cell and the major target is to find an algorithm that establishes the spatial connectivity. Here we stick to the following notation rule: Geometrical objects are denoted by capital letters, whereas numbers of geometrical objects are denoted by small letters. Spatially connected objects are accented using $\tilde{\cdot}$, whereas hierarchically connected objects are accented using $\cdot'$.

2.1. The cell type.

Let us introduce a few simple concepts, in order to fix the general structure of the geometrical objects we are going to consider. According to [17] a *polytope* is a compact and convex subset of $\mathbb{R}^n$, which is the intersection of a finite set of closed halfspaces of $\mathbb{R}^n$. Furthermore, the polytope is the convex hull of its *vertices* and if the polytope K is the intersection of the non-redundant set of closed halfspaces $H_1^+, H_2^+, \ldots, H_m^+$, then $F_i = H_i \cap K$ for $i = 1, 2, \ldots, m$ are the *facets* of K and $\partial K = \cup_{i=1}^m F_i$. We will use the term *face* instead of facet, since this is more common in our application background, but note that faces are different objects in the sense of [17]. If $X_1, X_2, \ldots, X_d \in \mathbb{R}^n$ are the vertices of the polytope K, then we call $\lambda \in \mathbb{R}^d$ an array of convex coefficients if $\lambda_i \geq 0$ for all i and $\sum_{i=1}^d \lambda_i = 1$. Then λ maps to a point $X \equiv \sum_{i=1}^d \lambda_i X_i \in K$. Note that in general for given $X \in K$ the corresponding array of convex coefficients is not unique. It is unique if K is an n–Simplex and in this case the array of convex coefficients represents the well-known barycentric coordinates.

#D	2
#N	3
#F	3
#NF	2

(a) triangle

#D	2
#N	4
#F	4
#NF	2

(b) quadrilateral

#D	3
#N	4
#F	4
#NF	3

(c) tetrahedron

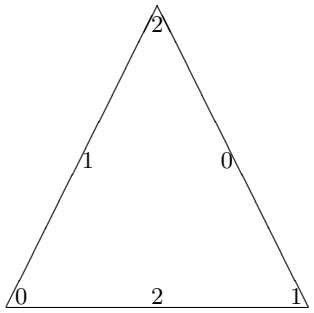
#D	3
#N	8
#F	6
#NF	4

(d) cuboid

#D	3
#N	6
#F	5
#NF	3 or 4

(e) prism

Fig. 2.1: The numbers $\#D, \#N, \#F$ and $\#NF$ for various cell types.



(a)

f	$n_0^F(f)$	$n_1^F(f)$
0	1	2
1	2	0
2	0	1

(b) n^F

Fig. 2.2: Table n^F for cell type triangle.

DEFINITION 2.1 (Cell, cell type). *Let $\#D, \#N, \#F, \#NF \in \mathbb{N}$ be given. Then we say that K is a cell of cell type $(\#D, \#N, \#F, \#NF)$, if*

(i) $K \subset \mathbb{R}^{\#D}$ is a polytope with $\#N$ vertices,

(ii) K is not contained in a hyperplane of $\mathbb{R}^{\#D}$,

(iii) K has $\#F$ faces and each face is a polytope with $\#NF$ vertices, where each vertex of each face is a vertex of K.

In Fig. 2.1 the numbers $\#D, \#N, \#F, \#NF$ are given for the cell types we will consider here. Next we introduce the local numbering of vertices and faces for a given cell type. Let $A_0, A_1, \ldots, A_{\#N-1}$ be the vertices of the cell K and define the local numbering of the vertices according to their subscript. Then the numbering of the faces in cell K is given by the table n^F, which is of size $\#F \times \#NF$ with values in $0, 1, \ldots, \#N - 1$, and tells us which vertices of K are the vertices of face f. In Fig. 2.2 we see this table for the cell type triangle and its illustration in the drawing, where the numbers of the (cell–) vertices and of the faces are indicated. To be more precise, let F be the face of K with face number f, then vertex j of F is vertex $n_j^F(f)$ of K. We also define the sets $n^F(f) \equiv \{n_0^F(f), n_1^F(f), \ldots, n_{\#NF-1}^F(f)\}$. For the cell type tetrahedron, the table n^F is given in Fig. 2.5.

We will discuss in detail the cell types triangle and tetrahedron in Sect. 2 and Sect. 3. Since the discussion for the cell types quadrilateral and cuboid is analogous and straight forward, we only display the corresponding tables in Appendix A, and restrict ourselves to a few comments for these cell types here.

2.2. The refinement rule and the cell identifier (ID). We further assign to each cell type a *refinement rule*, which divides a cell K into $\#C$ subcells, also denoted as *children* of K, which are of the same cell type as K. The number of subcells $\#C$ is given in Fig. 2.3 for the refinement rule we are going to employ for the respective cell type. We call K the *parent* of each of its children. Similarly we require that when a cell K is refined, each of its faces is subdivided into $\#CF$ subfaces, where each subface is a face of a child of K.

#C	4
#CF	2

(a) triangle

#C	4
#CF	2

(b) quadrilateral

#C	8
#CF	4

(c) tetrahedron

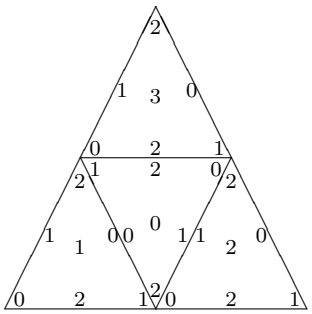
#C	8
#CF	4

(d) cuboid

#C	6
#CF	3 or 4

(e) prism

Fig. 2.3: The numbers $\#C$ and $\#CF$ for the refinement of various cell types.



(a)

c	$n_0^C(c)$	$n_1^C(c)$	$n_2^C(c)$
0	1,2	0,2	0,1
1	0	0,1	0,2
2	0,1	1	1,2
3	0,2	1,2	2

(b) n^C

Fig. 2.4: Table n^C for cell type triangle.

The general structure of the refinement rule is such that for any subcell K' of K each vertex of K' is the arithmetic mean of a collection of vertices of K. This is made explicit in the table n^C, which is of size $\#C \times \#N$. It is given in Fig. 2.4 for the cell type triangle and in Fig. 2.5 for the cell type tetrahedron. Furthermore, the drawing of Fig. 2.4 illustrates the table n^C for the cell type triangle. To be more precise, consider child $c \in \{0, 1, \ldots, \#C-1\}$ of K and assume that $A'_j \in \mathbb{R}^{\#D}$ is vertex $j \in \{0, 1, \ldots, \#N-1\}$ of the child cell. Then $n_j^C(c)$ tells us, which vertices of K have to be averaged to obtain A'_j. For example, according to the table of Fig. 2.4, if A_0, A_1, A_2 are the vertices of triangle K, then vertex 1 of subcell 0 is $\frac{1}{2}(A_0 + A_2)$, whereas vertex 0 of subcell 1 is A_0. This may be written in general as

$$A'_j \;=\; \frac{\sum_{k \in n_j^C(c)} A_k}{\sum_{k \in n_j^C(c)} 1}\,.$$

For the cell type triangle all entries of table n^C are illustrated in the drawing of Fig. 2.4, where the children of the triangle of Fig. 2.2 are displayed.

Table n^C can be applied recursively and thus defines a *refinement hierarchy*. The cell that contains all cells of the hierarchy is called the *base cell*; its faces are called the *base faces*. The children of the base cell are the cells of *level* 1, the children of all cells of level 1 are the cells of level 2 and so on. In the left picture of Fig. 2.6, we see the cells of level 2 for the base triangle given in Fig. 2.2. They are children of the cells of level 1 and their child numbers are indicated. We associate with each cell that can possibly be attained within the refinement hierarchy, the *path* of child numbers we have to pass through from level to level to reach that cell. In the right picture of Figure 2.6 we see the paths of the cells on level 2. Reading from right to left, the path gives the children passed through from lower to higher levels. The child numbers considered as entries in the path are called *digits*. Given a cell of level l we know that only the l digits associated with the lowest levels are of interest. We thus uniquely identify a cell by its level and a sequence of path digits. This information

f	$n_0^F(f)$	$n_1^F(f)$	$n_2^F(f)$
0	0	1	2
1	0	1	3
2	0	2	3
3	1	2	3

(a) n^F

c	$n_0^C(c)$	$n_1^C(c)$	$n_2^C(c)$	$n_3^C(c)$
0	1,2	0,2	0,1	0,3
1	1,3	0,3	1,2	0,1
2	2,3	1,2	0,3	0,2
3	0,3	2,3	1,3	1,2
4	0	0,1	0,2	0,3
5	0,1	1	1,2	1,3
6	0,2	1,2	2	2,3
7	0,3	1,3	2,3	3

(b) n^C

Fig. 2.5: Tables n^F and n^C for cell type tetrahedron.

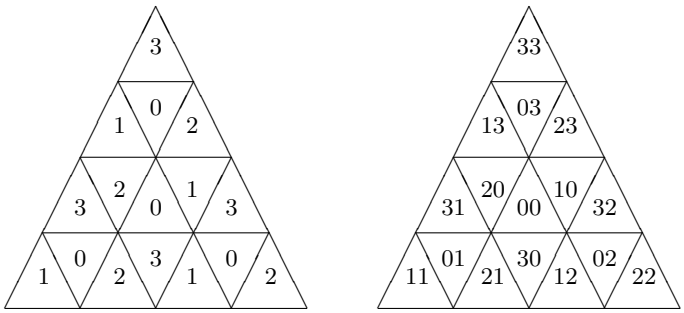


Fig. 2.6: Child numbers and paths associated with triangles for level 2

is collected in the *cell identifier* (𝕀𝔻):

$$\begin{aligned} \mathbb{ID} &\equiv (path, level), && && (2.1)\\ path &\equiv c_l \cdot \cdots \cdot c_3 c_2 c_1\,, && \text{with digits} && c_i \in \{0,1,\ldots,\#C-1\},\\ level &\equiv l\,, && \text{where} && l \in \{1,2,3,\ldots\}. \end{aligned}$$

Note that in (3.1) and (3.3), the cell identifier (2.1) will be extended by adding further information to the 𝕀𝔻. In the remainder of the current section we only consider the refinement hierarchy that belongs to a single base cell and the main purpose of the section is to solve Problem 2.2 within this refinement hierarchy.

PROBLEM 2.2 (Spatial connectivity). *Let an $\mathbb{ID} \equiv (c_l c_{l-1} \ldots c_2 c_1, l)$ and a face number f be given and let us identify $\mathbb{ID}$ with the cell it corresponds to. In case face f of cell $\mathbb{ID}$ is part of a base face, then we want to be provided with this information. Otherwise find $\tilde{\mathbb{ID}} \equiv (\tilde{c}_l \tilde{c}_{l-1} \ldots \tilde{c}_2 \tilde{c}_1, \tilde{l})$ and $\tilde{f}$ such that on the grid of level l, the neighbor of ID across its face f is $\tilde{\mathbb{ID}}$ and the neighbor of $\tilde{\mathbb{ID}}$ across its face $\tilde{f}$ is $\mathbb{ID}$.*

In Sect. 3 we consider a whole grid of base cells, the so-called coarse grid, and treat the refinement hierarchy associated with the coarse grid and also Problem 2.2.

2.3. Conditions imposed on the refinement rule. In this section we set up conditions on the refinement rule, which will enable us to solve Problem 2.2. These conditions will be formulated in Properties 2.3, 2.6, 2.8 and 2.12 below. These properties are easily checked for a given cell type and

f \ c	0	1	2	3
0	0	0	-1	-1
1	1	-1	1	-1
2	2	-1	-1	2

(a) triangle

f \ c	0	1	2	3	4	5	6	7
0	-1	2	1	0	-1	-1	-1	0
1	3	-1	1	0	-1	-1	1	-1
2	3	2	-1	0	-1	2	-1	-1
3	3	2	1	-1	3	-1	-1	-1

(b) tetrahedron

Fig. 2.7: Divide/insert–table $\tilde{f}(f,c)$ depending on face and child number.

its refinement rule, since these properties mostly concern the cells of level 1 and only in one instance (Property 2.12) the cells of level 2. If all properties can be verified, then useful conclusions, which concern all cells in the refinement hierarchy, will be seen to hold. This is detailed in the remainder of Sect. 2 and in Sect. 3. We will see that the refinement rules proposed for different cell types in Sect. 2.2 and Appendix A possess all these properties.

2.3.1. Divide and insert faces and the table $\tilde{f}(f,c)$. We can distinguish between two types of faces, namely the ones of type *divide* and the ones of type *insert*: Considering a cell of level 1 or higher, a divide face of the cell is a face that is contained in a face of the cell's parent. An insert face of the cell is a face that is not a divide face. The faces of the base cell are called the *base faces*. Thus any face, that is contained in a base face, is a divide face. We formulate the first property that our refinement rule should satisfy:

PROPERTY 2.3 (Conformity of insert faces of level 1). *For each insert face F of level 1, there exist two cells K, $\tilde{K}$ of level 1, such that $F = K \cap \tilde{K}$ and F is a face of K as well as a face of $\tilde{K}$.*

Due to the recursive application of the refinement rule, we obtain:

LEMMA 2.4 (Conformity of insert faces on all levels). *Given a cell type and a refinement rule according to Sect. 2.1 and 2.2 such that Property 2.3 is satisfied, then the conformity of insert faces on all levels is given: For each insert face F on level $l \geq 1$, there exist two cells K, $\tilde{K}$ of level l, such that $F = K \cap \tilde{K}$ and F is a face of K as well as a face of $\tilde{K}$.*

The validity of Property 2.3 can be read off from the drawing of Fig. 2.4 for the cell type triangle. In general it has to be checked in conjunction with tables n^F and n^C. It is easily checked, that this property also holds for the cell type tetrahedron.

Next, we can easily set up a table that tells us, if we are at a divide face, and if not, provides us with another valueable information: Consider first the triangles of level 1 in the drawing of Fig. 2.4. Given the child number c of a triangle and a face number f, we return the value $\tilde{f}(f,c) = -1$ if face f of the triangle is a divide face and in case the face is an insert face, according to Property 2.3 we know that the face also belongs to another triangle of level 1 and we return as $\tilde{f}(f,c)$ the face number with respect to the other triangle. The corresponding tables $\tilde{f}(f,c)$ are given for the cell types triangle and tetrahedron in Fig. 2.7. It is easily seen for the cells of level 1, that this table is well–defined if a cell type and a refinement rule is given according to Sect. 2.1, 2.2 and satisfies Property 2.3. And due to the recursive application of the refinement rule, it is also clear that the table is valid on all levels $l \geq 1$:

LEMMA 2.5 (Uniqueness of table $\tilde{f}(f,c)$). *Given a cell type and a refinement rule according to Sect. 2.1, 2.2 such that Property 2.3 is satisfied, then table $\tilde{f}(f,c)$ is well–defined throughout the refinement hierarchy and is already uniquely determined by level 1.*

2.3.2. Invariance of face numbers under refinement. From the drawings of Fig. 2.2 and Fig. 2.4 we can make another observation for the divide faces of level 1: Let K' be a triangle of level 1, a child of base cell K. Given a divide face of K', say with face number f', and assuming the face is contained in face f of K, then we have $f = f'$. Obviously, whenever this property holds on level 1, again due to the recursive application of the refinement rule, it will hold on all levels $l \geq 1$. We formulate the property that has to be verified and the conclusion separately:

PROPERTY 2.6 (Invariance of face numbers under refinement). *Given the base cell K and any one of its children K'. If face f' of K' is contained in face f of K, then we have $f = f'$.*

We immediately conclude:

LEMMA 2.7 (Invariance of face numbers under refinement). *Given a cell type and a refinement rule according to Sect. 2.1, 2.2 such that Property 2.6 is satisfied. Then if K is any cell in the refinement hierarchy, K' any of its children and face f' of K' is contained in face f of K, then we have $f = f'$.*

In fact, trying to ensure the validity of Property 2.6 in the case of tetrahedra led us to the special enumeration that can be found in tables n^C and n^F of Fig. 2.5. This enumeration is different to the ones available in the literature, and we will comment on this issue in detail in Sect. 2.5.

2.3.3. Invariant pattern of face division. Consider again the situation of Lemma 2.7, i.e. let K' be a child of cell K, F a face of K and let F' be a face of K' that is contained in F. Then we say that F' is a *child face* of F and F is the *parent face* of F'. Remember that F has $\#CF$ child faces with $\#CF$ given in the table of Fig. 2.3.

Now, if Lemma 2.7 holds and F is face f of K, then F' is also face f of K'. Each vertex of F' is a vertex of K', and thus due to table n^C each vertex of F' can be written as the arithmetic mean of a collection of vertices of K. In fact, since $F' \subset F$, the vertices of K to be averaged are vertices of F. Thus, with the aid of table n^C and n^F we can determine how each vertex of F' can be written as the arithmetic mean of a collection of vertices of F. In Property 2.8 we will impose an invariance on the procedure which determines the vertices of the face children of F from the vertices of F. We are now going to introduce the details for this:

Let $A_0, A_1, \ldots, A_{\#N-1}$ be the vertices of K, and $A'_0, A'_1, \ldots, A'_{\#N-1}$ the vertices of K', numbered according to the refinement rule of Sect. 2.1, 2.2. Let $\sigma \in \Pi_{\#NF}$, where Π_N is the group of permutations on a set of N elements, and define

$$\begin{aligned} E_i &= A_{n^F_{\sigma(i)}}(f) \quad \text{for } i \in \{0, 1, \ldots, \#NF - 1\}, \\ E'_i &= A'_{n^F_{\sigma(i)}}(f) \quad \text{for } i \in \{0, 1, \ldots, \#NF - 1\}. \end{aligned} \tag{2.2}$$

Then the points $E_0, E_1, \ldots, E_{\#NF-1}$, respectively $E'_0, E'_1, \ldots, E'_{\#NF-1}$, are the vertices of F, respectively F'. The E_i and E'_i can be found in table nc^F, which has $\#CF$ lines, see Fig. 2.8 for the cell types triangle and tetrahedron. Each line of table nc^F corresponds to one of the face children F' of F and tells us, which vertices from $E_0, E_1, \ldots, E_{\#N-1}$ have to be averaged to obtain the vertex E'_i of F'. Note that the $c'_i(\sigma, f)$ are not relevant yet and will be introduced shortly. Since triangles and quadrilaterals have the same geometrical objects as faces, we can employ the same permutations for both. Similarly tetrahedra and prisms both possess triangular faces; the quadrilateral faces of the prism will be treated in Appendix A.

We require that for a given cell type the table nc^F is unique and furthermore that it holds independent of the face number f of F and the permutation σ, i.e. independent of the numbering of the face vertices:

PROPERTY 2.8 (Pattern of vertices of face children). *There is a unique table nc^F, such that given any base face F and any $\sigma \in \Pi_{\#NF}$, each line of nc^F constitutes a rule that tells us for one child face F' of F, how the vertices E'_i of F' are given as arithmetic means of the vertices E_i of F. Here the E_i and the E'_i are given according to* (2.2).

Due to the recursive application of the refinement rule, we immediately obtain:

LEMMA 2.9 (Pattern of vertices of face children). *Given a cell type and a refinement rule according to Sect. 2.1, 2.2 such that Properties 2.6 and 2.8 are satisfied. Then the table nc^F of Property 2.8 holds for all faces F in the refinement hierarchy.*

The tables nc^F are illustrated in Fig. 2.9. Table nc^F is applied in the particular case of face $f = 1$ and face $f = 3$ of the cell type tetrahedron in Fig. 2.10. Here we have replaced E_i, E'_i from the right picture of Fig. 2.9 by the vertex numbers $n^F_{\sigma(i)}(f)$ from (2.2) with an appropriately chosen permutation σ.

To table nc^F we have added a column containing $c'_i(\sigma, f)$ and c'_i also occurs in Fig. 2.9, which has the following meaning: Let child face F' correspond to line $i \in \{0, 1, \ldots, \#CF\}$ of table nc^F. F also belongs to a particular child cell, with child number given by c'_i. How can we determine c'_i? For given $i \in \{0, 1, \ldots, \#CF\}$, σ, f, we can infer from (2.2) which of the A_i are averaged in each line of table nc^F and thus together with table n^C we can determine which child cell the child face associated with line i belongs to. To make the dependency on σ and f explicit we have written $c'_i(\sigma, f)$ in table nc^F. In the example of Fig. 2.10, the values c'_i are also indicated.

LEMMA 2.10 (Child numbers of cells belonging to face children).
For all $i \in \{0, 1, \ldots, \#CF\}$, $\sigma \in \Pi_{\#NF}$ and $f \in \{0, 1, \ldots, \#F\}$, the child number $c'_i(\sigma, f)$ is uniquely determined by combining (2.2) *and table n^C.*

As a last consequence of this subsection, combining Property 2.8 with Property 2.3 we obtain the conformity of uniformly refined grids:

LEMMA 2.11 (Conformity of uniformly refined grids). *Given a cell type and a refinement rule according to Sect. 2.1, 2.2 such that Properties 2.3, 2.6, 2.8 are satisfied. Let F be a face of level l in the refinement hierarchy. Then F is either contained in a base face, or $F = K \cap \tilde{K}$, where K, $\tilde{K}$ are two cells of level l in the refinement hierarchy and F is a face of K as well as a face of $\tilde{K}$.*

Proof: If F is not part of a base face, then it must be part of an insert face. If F is an insert face, then we are done according to Lemma 2.4.

What remains are the *inner divide faces*, i.e. those divide faces which are not contained in a base face, or expressed equivalently, those divide faces which are contained in an insert face. Let F' be the child face of the insert face $F = K \cap \tilde{K}$, where F, K, $\tilde{K}$ are all of level l. Then the vertices of F are at the same time vertices of K and $\tilde{K}$. When K and $\tilde{K}$ are refined, according to Lemma 2.9, F is divided as a face of K in the same way as F is divided as a face of $\tilde{K}$. One of the subfaces is F', and therefore, there exists a child K' of K and a child $\tilde{K}'$ of $\tilde{K}$ such that $F' = K' \cap \tilde{K}'$, where F', K', $\tilde{K}'$ are all of level $l + 1$.

This argument repeats, when F' is divided further in the refinement hierarchy, and thus for all divide faces, which are contained in the insert face F, the conformity condition holds. □

2.3.4. Table $\tilde{c}(\tilde{f}, f, c)$. In our effort to solve Problem 2.2, i.e. to find the neighbor cells of cell K across the faces of K, we have reached the following point: Given a face number f and the child number c of K we can already determine the number $\tilde{f}(f, c)$ of the face relative to the neighbor cell if we are at an insert face. If we are at a divide face, we can use the $\mathbb{ID}$ of K and Lemma 2.7 to do find $\tilde{f}$ as-well, which will be discussed in Sect. 2.4. One step further is to determine the child number

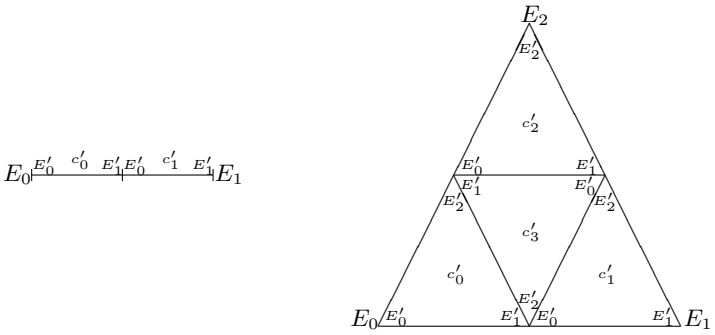
	nc^F	
	E'_0	E'_1
$c'_0(\sigma, f)$	E_0	E_0, E_1
$c'_1(\sigma, f)$	E_0, E_1	E_1

(a) Triangle or quadrilateral.

	nc^F		
	E'_0	E'_1	E'_2
$c'_0(\sigma, f)$	E_0	E_0, E_1	E_0, E_2
$c'_1(\sigma, f)$	E_0, E_1	E_1	E_1, E_2
$c'_2(\sigma, f)$	E_0, E_2	E_0, E_1	E_2
$c'_3(\sigma, f)$	E_1, E_2	E_0, E_2	E_0, E_1

(b) Tetrahedron or prism (triangular face).

Fig. 2.8: Tables nc^F and child numbers c'_i for vertices of face children.



(a) Face of a triangle. (b) Face of a tetrahedron.

Fig. 2.9: Unique pattern of vertex numbers.

of the neighbor cell when $\tilde{f}, f, c$ are given. This is established by the table $\tilde{c}(\tilde{f}, f, c)$, which will be discussed next:

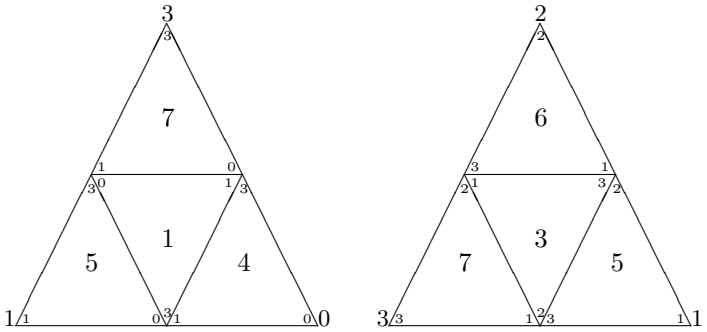
Let F be a face of level $l \geq 1$, which is not contained in a base face, and let Lemma 2.11 hold, such that $F = K \cap \tilde{K}$, and K, $\tilde{K}$ are two cells of level l in the refinement hierarchy. Let F be face f of K and face $\tilde{f}$ of $\tilde{K}$ and let K be child c of its parent cell and $\tilde{K}$ child $\tilde{c}$ of its parent cell. Then the return value of $\tilde{c}(\tilde{f}, f, c)$ is defined to be $\tilde{c}$. We demand that the following property holds.

PROPERTY 2.12. *Table $\tilde{c}(\tilde{f}, f, c)$ is uniquely defined for all inner divide faces of level 2.*

The validity of Property 2.12 for the cell types triangle and tetrahedron can be checked by employing tables n^F an n^C, while table $\tilde{c}(\tilde{f}, f, c)$ is given for cell types triangle and tetrahedron in Fig. 2.11 and Fig. 2.12. Now we can draw the following conclusion:

LEMMA 2.13. *Given a cell type and a refinement rule according to Sect. 2.1, 2.2 such that Property 2.3 is satisfied. Then table $\tilde{c}(\tilde{f}, f, c)$ is uniquely defined for all insert faces of level 1. If additionally Properties 2.6, 2.8, 2.12 hold, then table $\tilde{c}(\tilde{f}, f, c)$ is well-defined throughout the refinement hierarchy, and it is fully determined by the values found for the insert faces of level 1 and the inner divide faces of level 2.*

Proof: Let K be child c of the base cell, i.e. K is a cell of level 1, and f any face number. Of course, on level 1, the pair (f, c) occurs only once. If additionally $\tilde{f} = \tilde{f}(f, c) \neq -1$, i.e. face f of K is an insert face of level 1, then due to Property 2.3, $\tilde{c}(\tilde{f}, f, c)$ is well-defined. Due to the recursive application of the refinement rule, we immediately see, that $\tilde{c}(\tilde{f}, f, c)$ is given for the insert faces on all levels by the values $\tilde{c}(\tilde{f}, f, c)$ found on level 1.



(a) $f = 1$ (b) $f = 3$

Fig. 2.10: Unique pattern of vertex and child numbers for faces 1 and 3 of cell type tetrahedron.

f \ c	0	1	2	3
0	1	0	3	2
1	-1	-1	-1	-1
2	-1	-1	-1	-1

(a) $\tilde{c}(0, f, c)$

f \ c	0	1	2	3
0	-1	-1	-1	-1
1	2	3	0	1
2	-1	-1	-1	-1

(b) $\tilde{c}(1, f, c)$

f \ c	0	1	2	3
0	-1	-1	-1	-1
1	-1	-1	-1	-1
2	3	2	1	0

(c) $\tilde{c}(2, f, c)$

Fig. 2.11: Table $\tilde{c}(\tilde{f}, f, c)$ for cell type triangle.

It remains to check the inner divide faces. For divide faces we have $\tilde{f}(f, c) = -1$, and thus a pair (f, c) can never occur for both a divide face and an insert face. Thus the values $\tilde{c}(\tilde{f}, f, c)$ given for the inner divide faces of level 2 cannot conflict with the values given for insert faces.

It remains to check, that the values $\tilde{c}(\tilde{f}, f, c)$ given for the inner divide faces of level 2 cannot conflict with those values for other inner divide faces. On level 1 no inner divide faces occur. Let us have a closer look at the inner divide faces of level 2, which are exactly the child faces of insert faces of level 1: Let F be an insert face of level 1 with the two cells K and $\tilde{K}$ it belongs to, so that F is face f of K and face $\tilde{f}$ of $\tilde{K}$. Let $E_0, E_1, \ldots, E_{\#NF-1}$ be the vertices of F and $E_i = A_{n^F_{\sigma(i)}}(f) = \tilde{A}_{n^F_{\tilde{\sigma}(i)}}(\tilde{f})$ for $i \in \{0, 1, \ldots, \#NF - 1\}$, where the A_j are the vertices of K, the $\tilde{A}_j$ are the vertices of $\tilde{K}$ and $\sigma, \tilde{\sigma} \in \Pi_{\#NF}$ are chosen appropriately. Then each line of table nc^F represents a child face F' of F, where $F' = K' \cap \tilde{K}'$. Suppose we are in line $i \in \{0, 1, \ldots \#NF - 1\}$, so that K' is child $c'_i(\sigma, f)$ of K and $\tilde{K}'$ is child $c'_i(\tilde{\sigma}, \tilde{f})$ of $\tilde{K}$, which results in the entries $c'_i(\tilde{\sigma}, \tilde{f}) = \tilde{c}(\tilde{f}, f, c'_i(\sigma, f))$. Due to (2.2) for the vertices E'_i of F' we have $E'_i = A'_{n^F_{\sigma(i)}}(f) = \tilde{A}'_{n^F_{\tilde{\sigma}(i)}}(\tilde{f})$ for $i \in \{0, 1, \ldots, \#NF - 1\}$. In particular the same permutations σ, $\tilde{\sigma}$ and face numbers f, $\tilde{f}$ occur, and thus on F' the same situation occurs as on F, i.e. the same table entries $c'_i(\tilde{\sigma}, \tilde{f}) = \tilde{c}(\tilde{f}, f, c'_i(\sigma, f))$ are reproduced. This argument repeats when refining further, and thus on all faces which are part of a divide face of level 2, the same entries occur in table $\tilde{c}(\tilde{f}, f, c)$.

Due to the recursive application of the refinement rule, it is also clear that for any inner divide face which is the child face of an insert face, the same entries in $\tilde{c}(\tilde{f}, f, c)$ occur as for the inner divide faces of level 2. Again, when refining the inner divide face, we can use the argument from above, and we finally obtain that for all divide faces the entries in table $\tilde{c}(\tilde{f}, f, c)$ are equal to those valid for the inner divide faces of level 2. □

2.4. Connectivity. Given an $\mathbb{ID}$, to determine the $\mathbb{ID}$s in the hierarchical connectivity is an easy task: The parent is given by $\mathbb{ID}_{parent} = (c_{l-1} \ldots c_2 c_1, l - 1)$ and the j–th child by $\mathbb{ID}_{child\ j} =$

f \ c	0	1	2	3	4	5	6	7
0	0	-1	-1	7	4	6	5	3
1	-1	0	-1	2	6	4	-1	5
2	-1	-1	0	1	5	-1	4	6
3	-1	-1	-1	-1	-1	-1	-1	-1

(a) $\tilde{c}(0, f, c)$

f \ c	0	1	2	3	4	5	6	7
0	1	-1	3	-1	5	7	4	-1
1	-1	1	6	-1	7	5	2	4
2	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	0	1	-1	4	7	5

(b) $\tilde{c}(1, f, c)$

f \ c	0	1	2	3	4	5	6	7
0	2	3	-1	-1	6	4	7	-1
1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	5	2	-1	7	1	6	4
3	-1	0	-1	2	-1	7	4	6

(c) $\tilde{c}(2, f, c)$

f \ c	0	1	2	3	4	5	6	7
0	-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	-1	-1	5	7	-1	6
2	1	-1	3	-1	6	-1	7	5
3	4	-1	-1	3	0	6	5	7

(d) $\tilde{c}(3, f, c)$

Fig. 2.12: Table $\tilde{c}(\tilde{f}, f, c)$ for cell type tetrahedron.

$(jc_lc_{l-1}\dots c_3c_2c_1, l+1)$ for $j \in \{0, 1, \dots, \#C-1\}$. Problem 2.2, which concerns the spatial connectivity, is more involved. Assuming that all properties formulated so far are satisfied by the refinement rule, Problem 2.2 can be solved as follows:

Let the assumptions of Lemma 2.13 hold. If $\tilde{f}(f, c_l) = -1$ then face f of $\mathbb{ID}$ is a divide face and due to Lemma 2.7 it is a child of face f of $(c_{l-1}\dots c_2c_1, l-1)$. If $\tilde{f}(f, c_{l-1}) = -1$ then face f of $(c_{l-1}\dots c_2c_1, l-1)$ is a divide face and due to Lemma 2.7 it is a child of face f of $(c_{l-2}\dots c_2c_1, l-2)$ and so on. We finally have that if $\tilde{f}(f, c_j) = -1$ for $j = l, l-1, \dots, 1$, then face f of $\mathbb{ID}$ is contained in a base face. Otherwise, let $i \le l$ be the largest index with $\tilde{f}(f, c_i) \neq -1$. Then we know that face f of $(c_i\dots c_2c_1, i)$ is an insert face and it contains face f of $\mathbb{ID}$. Furthermore, we know that face f of $(c_i\dots c_2c_1, i)$ is also face $\tilde{f}$ of $(\tilde{c}_i, c_{i-1}\dots c_2c_1, i)$, where $\tilde{f} = \tilde{f}(f, c_i)$ and $\tilde{c}_i = \tilde{c}(\tilde{f}, f, c_i)$, i.e. the cells $(c_i\dots c_2c_1, i)$ and $(\tilde{c}_i, c_{i-1}\dots c_2c_1, i)$ are neighbor cells. Similarly we find, that $(c_j\dots c_2c_1, j)$ and $(\tilde{c}_j, \tilde{c}_{j-1}, \dots, \tilde{c}_i, c_{i-1}\dots c_2c_1, j)$ are neighbors for $j \ge i$, where $\tilde{c}_k = \tilde{c}(\tilde{f}, f, c_k)$ for $k \ge i$ and their common face is face f of $(c_j\dots c_2c_1, j)$ or likewise face $\tilde{f}$ of $(\tilde{c}_j, \tilde{c}_{j-1}, \dots, \tilde{c}_i, c_{i-1}\dots c_2c_1, j)$. Thus we have solved Problem 2.2, and display the algorithm that finds the solution in Algorithm 1.

LEMMA 2.14. *Given a cell type and a refinement rule according to Sect. 2.1, 2.2 such that all properties of Sect. 2.3 hold, then Problem2.2 is solved by Algorithm 1.*

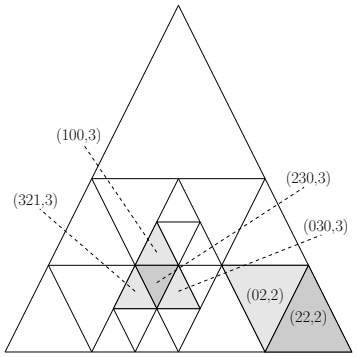
Because of looping twice over the path, the complexity of Algorithm 1 is proportional to the level present in the adaptive grid, so it is of the order of the maximal level of cells in the grid. But the algorithm can also be formulated to find the $\mathbb{ID}$s of neighboring cells in constant time, see Sect. 4.2. As an example, the neighbor triangles of triangle $\mathbb{ID} = (230, 3)$, respectively $\mathbb{ID} = (22, 2)$, are given in Fig. 2.13.

2.5. Congruence classes. Two cells K and $\tilde{K}$ are said to be *congruent*, if there exists a scaling factor $c > 0$, a translation vector $r \in \mathbb{R}^{\#D}$ and an orthogonal matrix Q, such that $K = r + cQ\tilde{K}$.

We easily see that all cells appearing in the refinement hierarchy of a base triangle are congruent to the base triangle. For the cell type tetrahedron this is not the case and the analysis of this issue is quite involved. The issue was analyzed by Bey [4, 5], where the table n^C of Fig. 2.14 was employed. The result being that amongst the base tetrahedron and all the cells in its refinement hierarchy, there are at most three congruence classes. The finiteness of congruence classes ensures the stability of the hierarchy, which means that all tetrahedra are *shape-regular*, i.e. there exists $\varrho > 0$ such that $\varrho\, r_K \ge h_K$ for all tetrahedra K in the refinement hierarchy. Here r_K is the radius of the smallest

Algorithm 1 Neighboring algorithm within one base cell: Given $\mathbb{ID} \equiv (c_l c_{l-1} \dots c_2 c_1, l)$ and face number f this algorithm solves Problem 2.2, i.e. it determines $\tilde{\mathbb{ID}} \equiv (\tilde{c}_l \tilde{c}_{l-1} \dots \tilde{c}_2 \tilde{c}_1, \tilde{l})$ and $\tilde{f}$ where $\tilde{f} == -1$ means that we are at a base face. The determination of o will be explained in Sect. 2.7.

```
1: i = l + 1;
2: l̃ = l;
3: f̃ = −1;
4: while (f̃ == −1 and i > 1) do                ▷ Search for insert face
5:     i = i − 1;
6:     f̃ = f̃(f, c_i);
7: end while
8: if f̃ ≠ −1 then                              ▷ Insert face found
9:     o = o(f, c_i);                          ▷ Look up orientation of neighbor cell in table o(f, c)
10:    for j = 1, ..., i − 1 do                ▷ Copy digits c_j for j = 1, ..., i − 1 from input path
11:        c̃_j = c_j;
12:    end for
13:    for j = i, ..., l do                    ▷ Apply table c̃(f̃, f, c) to digits c_j for j = i, ..., l
14:        c̃_j = c̃(f̃, f, c_j);
15:    end for
16: end if
```



$\mathbb{ID}$	f	$\tilde{\mathbb{ID}}$	$\tilde{f}$
(230,3)	0	(321, 3)	0
	1	(030, 3)	1
	2	(100, 3)	2
(22, 2)	0	–	-1
	1	(02, 2)	1
	2	–	-1

Fig. 2.13: The neighbor triangles of triangle $\mathbb{ID} = (230, 3)$ and $\mathbb{ID} = (22, 2)$.

ball contained in K and h_K is the diameter of K. This shape-regularity is an important condition necessary for the convergence of Discontinuous Galerkin or Finite Element Methods.

The reason why we employed a different numbering is that the numbering of Bey does not satisfy Property 2.6, which has been made use of extensively above to solve Problem 2.2 and will further be needed in Sect. 2.7 and in Sect. 3.

Using a computer algebra system, we have determined the number of congruence classes for the refinement as given in Fig. 2.5, which also results in three congruence classes. If the base tetrahedron is given with vertices (A_0, A_1, A_2, A_3), then the three tetrahedra with vertices $(0, A_1 - A_0, A_2 - A_0, A_3 - A_0)$, $(0, A_1 - A_0, A_1 - A_2, A_3 - A_2)$ and $(0, A_1 - A_0, A_3 - A_2, A_3 - A_0)$ are representatives of the three congruence classes appearing in the refinement hierarchy of the base tetrahedron.

In order to exploit simple scalings to determine stiffness or mass matrices, one has to find appropriate equivalence relations. E.g. if we need to consider those tetrahedra which can be mapped onto each other by translation and positive scaling only, then the three congruence classes given above split further and create at most 24 equivalence classes. In comparison, Bey's refinement results in at most 6 equivalence classes. For a given base tetrahedron, the equivalence class a subtetrahedron belongs to can be determined from its path using a finite state machine, a well-known concept from

c	$n_0^C(c)$	$n_1^C(c)$	$n_2^C(c)$	$n_3^C(c)$
0	0	0,1	0,2	0,3
1	0,1	1	1,2	1,3
2	0,2	1,2	2	2,3
3	0,3	1,3	2,3	3
4	0,1	0,2	0,3	1,3
5	0,1	0,2	1,2	1,3
6	0,2	0,3	1,3	2,3
7	0,2	1,2	1,3	2,3

Fig. 2.14: n^C for Bey's refinement rule [4].

automata theory, see [15, 22]. In many instances larger equivalence classes suffice, e.g. if the volume of a tetrahedron is needed, it can be precomputed for the three congruence classes given above, the congruence class and level of the tetrahedron is determined and the volume is given by an appropriate scaling.

2.6. Neighbor orientation. Assume that all properties of Sect. 2.3 are satisfied. So far we have only discussed, how Algorithm 1 yields the neighbor cell $\tilde{\mathbb{D}}$ and face number $\tilde{f}$. We now ask, how neighboring cells meet vertex-wise at a face. For this purpose let us consider the following situation: Let K and $\tilde{K}$ be two cells of the same level and face $F = K \cap \tilde{K}$ such that F is face f of K and face $\tilde{f}$ of $\tilde{K}$. Furthermore let $A_0, A_1, \ldots, A_{\#N-1}$, respectively $\tilde{A}_0, \tilde{A}_1, \ldots, \tilde{A}_{\#N-1}$, be the vertices of K, respectively $\tilde{K}$, given in the order defined by table n^C. In order to establish a quadrature rule for surface integrals in Sect. 2.7, the *orientation* of the two adjacent cells K and $\tilde{K}$ at face F has to be identified, i.e. we have to determine which vertices of K and $\tilde{K}$ are identical points in physical space. This information can be encoded in a permutation $\pi \in \Pi_{\#NF}$ such that

$$A_{n_j^F(f)} \;=\; \tilde{A}_{n^F_{\pi(j)}(\tilde{f})} \qquad \text{for all } j \in \{0, 1, \ldots, \#NF - 1\}. \tag{2.3}$$

We enumerate all possible permutations $\Pi_{\#NF} \equiv \{\pi_0, \pi_1, \ldots, \pi_{\#NF!-1}\}$. Assuming that the child number of K is c, then $o(f, c)$ is defined as the number of the permutation appearing in (2.3), i.e. $\pi = \pi_{o(f,c)}$. We first find the entries $o(f, c)$ for all insert faces of level 1. These entries are uniquely defined, only occur where table $\tilde{f}(f, c)$ has entries $\neq -1$, and are given in Fig. 2.15. The numbering of the permutations π_k is fixed in Fig. 2.16.

Due to Lemmas 2.7 and 2.9, table $o(f, c)$ remains valid in (2.3), if instead of F we consider any child of F. Again due to Lemmas 2.7 and 2.9, refining the face even further, π will remain the correct permutation in (2.3). Now consider any inner face F in the refinement hierarchy, for which we assume the situation given at the beginning of the section, and we ask again, what is the correct permutation in (2.3)? Then we have to coarsen F until an insert face occurs. If this it occurs at level i, then taking the path digit c_i of K, (2.3) holds with $\pi = \pi_{o(f,c_i)}$. This is how the permutation number $o = o(f, c_i)$ is determined in line 9 of Algorithm 1.

2.7. Face quadrature. When assembling the discrete equations of a DG method, one major task is to calculate surface integrals on the faces of the grid, which requires the evaluation of quadrature formulas. Typically for a face F, which is the common face of two cells K and $\tilde{K}$, such a surface integral and its approximation have the form

$$\int_F g(x, v(x), \tilde{v}(x))\, ds \;\approx\; |F| \sum_{i=0}^{\#QF-1} \omega_i\, g(X_i, v(X_i), \tilde{v}(X_i)). \tag{2.4}$$

Here v, respectively $\tilde{v}$, are continuous functions defined on K, respectively $\tilde{K}$, $\#QF$ is the number of face quadrature points, ω_i are the quadrature weights and X_i are the quadrature nodes. In most

f \ c	0	1	2	3
0	1	1	-1	-1
1	1	-1	1	-1
2	1	-1	-1	1

(a) triangle

f \ c	0	1	2	3	4	5	6	7
0	-1	3	2	1	-1	-1	-1	1
1	1	-1	5	4	-1	-1	5	-1
2	2	5	-1	3	-1	5	-1	-1
3	3	4	1	-1	3	-1	-1	-1

(b) tetrahedron

Fig. 2.15: Orientation table $o(f, c)$ for cell types triangle and tetrahedron.

	0	1
π_0	0	1
π_1	1	0

(a) Triangle or quadrilateral.

	0	1	2
π_0	0	1	2
π_1	0	2	1
π_2	1	2	0

	0	1	2
π_3	1	0	2
π_4	2	0	1
π_5	2	1	0

(b) Tetrahedron or prism (triangular face).

Fig. 2.16: Orientation permutations π_k for various cell types.

cases v, respectively $\tilde{v}$, are linear combinations of the DG shape functions.

In order to reduce function evaluations, one typically stores the shape function values at the face quadrature points on the *reference cell*. Working with a reference cell or *reference element* is a well-known technique developed in the Finite Element Method, see [9]. Thus $v(X_i)$ is provided via the numbering of the face quadrature points w.r.t. cell K, while $\tilde{v}(X_i)$ is provided via the numbering of the face quadrature points w.r.t. cell $\tilde{K}$.

Therefore we have to know which face quadrature points of K and $\tilde{K}$ coincide in physical space and in particular, we have to ensure in the first place that face quadrature points of K and $\tilde{K}$ do coincide at all. Thus we consider face quadrature formulas of the following form:

PROPERTY 2.15 (Face quadrature). *Let weights $\omega_i \in \mathbb{R}$ and arrays of convex coefficients $\eta^i \in \mathbb{R}^{\#NF}$ be given for $i \in \{0, 1, \ldots, \#QF - 1\}$, such that whenever $E_0, E_1, \ldots, E_{\#NF-1}$ are the vertices of a face F, we define the quadrature nodes to be $X_i \equiv \sum_{j=0}^{\#NF-1} \eta_j^i E_j$ for $i \in \{0, 1, \ldots, \#QF - 1\}$. Further for all permutations in $\Pi_{\#NF} \equiv \{\pi_0, \pi_1, \ldots, \pi_{\#NF!-1}\}$, we have corresponding permutations $\sigma_0, \sigma_1, \ldots, \sigma_{\#NF!-1} \in S_{\#QF}$, such that $X_{\sigma_k(i)} = \sum_{j=0}^{\#NF-1} \eta_j^i E_{\pi_k(j)}$ and $\omega_{\sigma_k(i)} = \omega_i$ for all $k \in \{0, 1, \ldots, \#NF! - 1\}$ and $i \in \{0, 1, \ldots, \#QF - 1\}$.*

The quadrature nodes on face f of a cell are denoted by X_i^f for $i \in \{0, 1, \ldots, \#QF - 1\}$. In Fig 2.17 we display the quadrature nodes when employing Gaussian quadrature with $\#QF = 2$ along the faces of triangles. In the right picture, quadrature nodes are shown and the different numberings they possess with respect to the adjacent cells K and $\tilde{K}$. In fact in our implementation, we allow the user to define quadrature formulas by providing $\omega_i, \eta^i, \sigma_k$ satisfying Property 2.15.

Since the permutation number o that we have found in Sect. 2.6 tells us which vertices of K and $\tilde{K}$ coincide on face F, and since the quadrature points of K and $\tilde{K}$ are defined in terms of these vertices according to Property 2.15, the permutation number o can be used for the evaluation of the

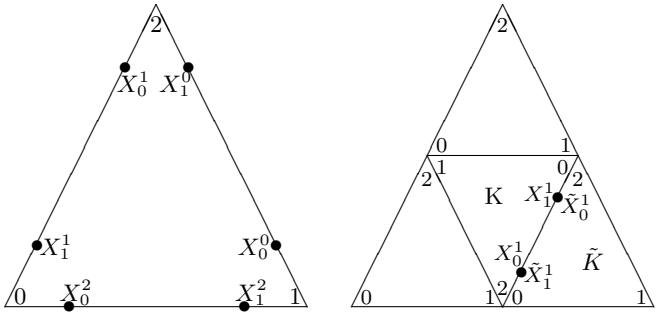


Fig. 2.17: Quadrature nodes.

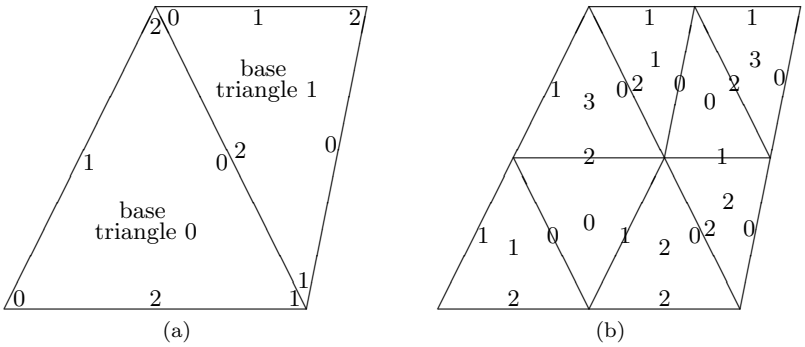


Fig. 3.1: Neighboring base triangles (a) and their uniform refinement of level 1 (b).

integral (2.4). This yields

$$\int_F g(x, v(x), \tilde{v}(x))\, ds \;\approx\; |F| \sum_{i=0}^{\#QF-1} \omega_i\, g(X_i^f, v(X_i^f), \tilde{v}(\tilde{X}^{\tilde{f}}_{\sigma_o(i)}))\,. \tag{2.5}$$

3. Adaptive refinement of a coarse grid. Assume we have given a coarse grid of a domain, obtained by an automatic mesh generator or by hand, such that no hanging vertices occur. Furthermore, let a problem be given that we assume to be solved on a grid that is obtained by refinements of the coarse grid. Typically this refined grid consists of a lot more cells than the coarse grid. We will consider each cell of the coarse grid as a base cell in the sense of Sect. 2 and extend the methods of Sect. 2. We assume that all base cells are of the same cell type, which possesses a refinement rule according to Sect. 2.1, 2.2 and satisfies all properties of Sect. 2.3. We will change this setting only in Sect. 3.4 when discussing hybrid grids, i.e. cases where the coarse grid contains base cells of different cell types.

3.1. The coarse grid and its connectivity. We are now going to specify how we define the spatial connectivity of the coarse grid. We assume that the coarse grid consists of $\#B$ base cells. Let F be face $f \in \{0, 1, \ldots, \#F - 1\}$ of base cell K and let K have base cell number $b \in \{0, 1, 2, \ldots, \#B - 1\}$. The spatial connectivity of the coarse grid is provided by assigning to each pair (b, f) values $\tilde{b}, \tilde{f}, o, \tilde{c}(0), \tilde{c}(1), \ldots, \tilde{c}(\#C - 1)$, which will be explained in the following. This amounts to storing $\#B \cdot \#F \cdot (\#C + 3)$ values.

We set $\tilde{f} = -1$, if F is on the boundary of the grid and assign in that case to $\tilde{b}$ a negative value, which serves as a boundary marker. Typically this boundary marker indicates the type of boundary

condition to be imposed, if we use the grid to solve a PDE with boundary conditions. Otherwise, i.e. in the case that F is not on the boundary, $\tilde{f}$ and $\tilde{b}$ are assigned such that $\tilde{K}$ is another base cell satisfying $F = K \cap \tilde{K}$ and F is face $\tilde{f}$ of $\tilde{K}$ and $\tilde{K}$ has base cell number $\tilde{b}$. Assuming further that o is defined such that (2.3) is satisfied with $\pi = \pi_o$, where the permutation π_o is given in Property 2.15. The values $o, \tilde{c}(0), \tilde{c}(1), \ldots, \tilde{c}(\#C-1)$ only have to be assigned if F is not on the boundary. What remains is to assign the values $\tilde{c}(0), \tilde{c}(1), \ldots, \tilde{c}(\#C-1)$, when $F, K, \tilde{K}, f, \tilde{f}$ are given as above. According to table c^F, compare Lemma 2.10, let K'_j be child $c^F_j(f)$ of K and let $\tilde{K}'_j$ be child $c^F_j(\tilde{f})$ of $\tilde{K}$. Since the vertices of K and $\tilde{K}$ are numbered independent of each other, we only know that there is a permutation $\sigma \in S_{\#CF}$, such that child $c^F_j(f)$ of K is the neighbor of child $c^F_{\sigma(j)}(\tilde{f})$ of $\tilde{K}$ for all $j \in \{0, 1, \ldots, \#CF-1\}$. We set $\tilde{c}(c^F_j(f)) \equiv c^F_{\sigma(j)}(\tilde{f})$ for $j \in \{0, 1, \ldots, \#CF-1\}$. In fact this defines only $\#CF$ values of $\tilde{c}(0), \tilde{c}(1), \ldots, \tilde{c}(\#C-1)$. The undefined values will not be needed in the sequel. When setting up the connectivity of the coarse grid, the entries $\tilde{c}(c^F_j(f))$ and the values $\tilde{b}, \tilde{f}, o$ have to be determined from the information given by the coarse mesh generator.

Since due to Lemma 2.10 the patterns of the child numbers of K and of $\tilde{K}$ replicate on any face in the refinement hierarchy that is contained in the base face F, we know that across these faces the child numbers of neighbor cells remain to be $c^F_j(f)$ and $c^F_{\sigma(j)}(\tilde{f})$ for $j \in \{0, 1, \ldots, \#CF-1\}$. This is an important observation that will be used in Sect. 3.2.

On the left of Fig. 3.1 we see a coarse triangulation consisting of two base triangles. In this case, for the pair $(b, f) = (0, 0)$ we have $\tilde{b} = 1$, $\tilde{f} = 2$, $o = 1$, and from the uniform refinement of level 1, shown on the right of Fig. 3.1 we obtain $\tilde{c}(2) = 2$ and $\tilde{c}(3) = 1$.

3.2. Uniform refinement of the coarse grid and connectivity. If we refine any base cell of the coarse grid in the manner of Sect. 2, then each cell that can possibly be attained by refinement is uniquely defined by the extended cell identifier

$$\mathbb{ID} = (path, level, basecell). \tag{3.1}$$

This means the considered cell is in the base cell with number *basecell* and is obtained inside this base cell via *path* and *level*. Note that in the following $\mathbb{ID}$ denotes the cell identifier in the sense of (3.1), until we extend it for a last time in (3.3). Since the hierarchical connectivity is not influenced by the coarse grid, parent and children $\mathbb{ID}$s are obtained as in Sect. 2.4, leaving the *basecell* number unchanged.

We consider again the question of the spatial connectivity as formulated in Problem 2.2, but now on uniform refinements of the coarse grid. The problem is thus slightly varied, since neighboring cells may belong to different base cells. Given a cell's $\mathbb{ID} = (path, level, basecell)$ and a face number f we search for the neighbor $\tilde{\mathbb{ID}}$ on the same level and the corresponding $\tilde{f}$. On the right of Fig. 3.1 we see how the base triangles are uniformly refined to level 1 and the face numbers are indicated. On the left of Fig. 3.2 we see how the base triangles from Fig. 3.1 are uniformly refined to level 2, with the paths of all triangles indicated. Excerpting $(path, level)$ from $\mathbb{ID}$ and applying Algorithm 1 to $(path, level)$ and the given face number f, we get either the information, that the face is part of a base face or we obtain the path of the neighboring cell on level l, which then lies in the same base cell. In the former case, from the spatial connectivity of the coarse grid we get the number of the base cell in which we can find our neighbor cell, or the information that we are at the boundary of the grid. What remains to be discussed is: If face f of cell $\mathbb{ID}$ is part of a base face and the neighboring base cell exists, what is the path of $\tilde{\mathbb{ID}}$. According to the discussion in Sect. 3.1, from the coarse connectivity we also know the array $\tilde{c}$ for face f of the given base cell. Thus we obtain the child digits $\tilde{c}_l, \ldots, \tilde{c}_1$ in the path of $\tilde{\mathbb{ID}}$ from the child digits $c_l, \ldots, c_1$ in the path of $\mathbb{ID}$ simply by setting $\tilde{c}_j = \tilde{c}(c_j)$. The full algorithm to determine the spatial connectivity within the uniformly refined grid is given in Algorithm 2. Using the permutation number o determined in Algorithm 2, the face quadrature formulae (2.5) holds in analogy to Sect. 2.7.

Algorithm 2 Neighboring algorithm with coarse connectivity: Given $\mathbb{ID} \equiv (c_l c_{l-1} \dots c_2 c_1, l, b)$ and face number f the following algorithm solves Problem 2.2, i.e. it determines $\tilde{\mathbb{ID}} \equiv (\tilde{c}_l \tilde{c}_{l-1} \dots \tilde{c}_2 \tilde{c}_1, \tilde{l}, \tilde{b})$ and $\tilde{f}$, where $\tilde{f} == -1$, means that we are at the grid boundary. Additionally the permutation number o is determined.

```
Excerpt (path, level) ≡ (c_l c_{l-1} ... c_2 c_1, l) from ID
Apply Algorithm 1 to obtain (c̃_l c̃_{l-1} ... c̃_2 c̃_1, l̃), f̃ and o from (c_l c_{l-1} ... c_2 c_1, l) and f;
if f̃ ≠ −1 then
    b̃ = b;
else
    Get b̃, f̃, o from b, f due to coarse connectivity;
    if f̃ ≠ −1 then
        Get c̃ from b, f due to coarse connectivity;
        for j=1,...,l do
            c̃_j = c̃(c_j);
        end for
    end if
end if
```

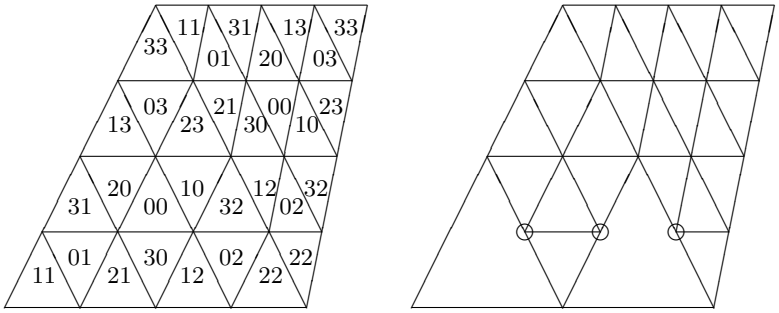


Fig. 3.2: Two base cells of Fig. 3.1 uniformly refined to level 2 with paths indicated (left), and refined to a simply graded grid with hanging vertices encircled (right).

3.3. Simply graded grids. The use of uniformly refined grids is inefficient in many applications. To enable local refinements we consider *simply graded grids* which are refinements of the coarse grid fullfilling the following condition: If two cells are adjacent across a face then their level can differ at most by 1. Due to the structure of our refinement rules, in a simply graded grid, a cell of level l can have $\#CF$ neighboring cells across one face, which are all of level $l+1$. On such a face the cells of the higher level possess vertices that are not vertices for the cell of the lower level, so–called *hanging vertices*, compare the encircled vertices in Fig. 3.2. In Sect. 4.5.1 we will give an example for an algorithm, which ensures that grids are simply graded.

Given $\mathbb{ID} \equiv (c_l c_{l-1} \dots c_2 c_1, l, b)$ and f, what are the neighbors $\tilde{\mathbb{ID}} \equiv (\tilde{c}_l \tilde{c}_{l-1} \dots \tilde{c}_2 \tilde{c}_1, \tilde{l}, \tilde{b})$ and $\tilde{f}$ in a simply graded grid? In Sect. 4.5.2 we will see that for the applications we consider here, it suffices to consider the case that $\tilde{\mathbb{ID}}$ is not of higher level than $\mathbb{ID}$, i.e. $\tilde{l} \in \{l, l-1\}$. Then according to Algorithm 2 from $\mathbb{ID}$ we first find the neighbor $\tilde{\mathbb{ID}} \equiv (\tilde{c}_l \tilde{c}_{l-1} \dots \tilde{c}_2 \tilde{c}_1, l, \tilde{b})$, which is of the same level, and the corresponding $\tilde{f}$ and o. We will see in Sect. 4.4 how we can check, if $\tilde{\mathbb{ID}}$ is actually a cell in the graded grid. If that cell exists, we are done. If it does not exist, $\tilde{\mathbb{ID}}$ is replaced by its parent $(\tilde{c}_{l-1} \dots \tilde{c}_2 \tilde{c}_1, l-1, \tilde{b})$. In this latter case, we have to reconsider the face quadrature (2.5): Note that from our initial application of Algorithm 2 we still have the face number $\tilde{f}$, the permutation number o and the digit $\tilde{c}_l$ to our disposal. Let cell K, respectively $\tilde{K}$, be the cells corresponding to $\mathbb{ID}$, respectively $\tilde{\mathbb{ID}} \equiv (\tilde{c}_{l-1} \dots \tilde{c}_2 \tilde{c}_1, l-1, \tilde{b})$. Then $F = K \cap \tilde{K}$ is face f of K and a child face of face

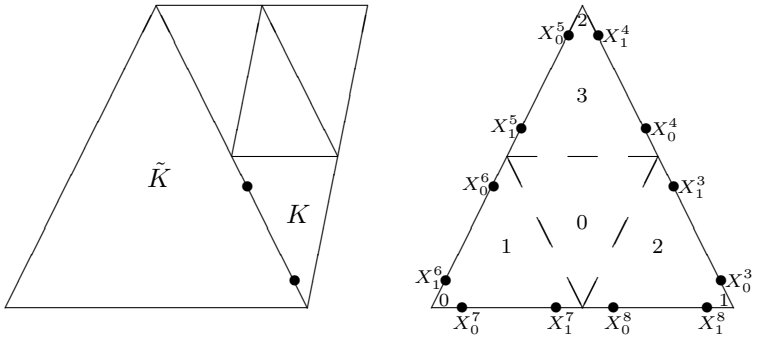


Fig. 3.3: Additional quadrature points.

$\tilde{f}$ of $\tilde{K}$, compare the left picture of Fig. 3.3. Note that additional face quadrature points for $\tilde{K}$ are required. Namely, apart from the face quadrature points $X_i^e,\ e \in \{0, ..., \#F - 1\}$, given in the left picture of Fig. 2.17, we have to add the face quadrature points given in the right picture of Fig. 3.3, which we denote by $X_i^e,\ e \in \{\#F, \#F + 1, \ldots, \#F \cdot (\#CF + 1) - 1\}$. Here we have numbered the child faces of the faces of K from $\#F$ to $\#F \cdot (\#CF + 1) - 1$. One easily sees that these child faces are also uniquely characterized by providing a cell child number c and a face number f, so that the corresponding child face is face f of the child c of K. This results in a table $e(f, c)$, which determines $e \in \{\#F, \#F + 1, \ldots, \#F \cdot (\#CF + 1) - 1\}$ from the corresponding f and c. The details are not given here. Now, using $\tilde{e} = e(\tilde{f}, \tilde{c}_l)$, the face quadrature (2.5) looks as follows:

$$\int_F g(x, v(x), \tilde{v}(x))\, ds \;\approx\; |F| \sum_{i=0}^{\#QF-1} \omega_i\, g(X_i^f, v(X_i^f), \tilde{v}(\tilde{X}^{\tilde{e}}_{\sigma_o(i)}))\,. \tag{3.2}$$

3.4. Hybrid grids. We have also implemented the case of hybrid grids. Namely, in 2D our coarse grid can consist of triangles and quadrilaterals, whereas in 3D the coarse grid can consist of tetrahedra, cuboids and prisms. For the cell types quadrilateral, cuboid and prism, we can set up the same tables that we have introduced for the cell types triangle and tetrahedron in the previous sections. Almost all arguments used so far carry over to the cell types quadrilateral, cuboid and prism. For cuboids the only change is that not all permutations $\pi \in \Pi_{\#NF}$ are allowed in (2.3). Here for a face F of a cuboid only those permutations of the vertices of F are allowed which correspond to rigid body motions of F, compare Fig. A.3. This is no restriction in practice, since the vertices of the base cuboids are naturally numbered such that those permutations do not occur. Furthermore, prisms are constructed as a tensor product of triangles and quadrilaterals. For details, see Appendix A.

In order to handle hybrid grids we partition our coarse grid into several *blocks*, where each block consists of cells of the same cell type. Accordingly we have to extend the ID again. Thus our cells are finally identified by

$$\mathrm{ID} \equiv (\mathit{path}, \mathit{level}, \mathit{basecell}, \mathit{block}, \mathit{type}, \mathit{flags}), \tag{3.3}$$

where in addition to the information stored in the previous cell identifier, the block number and the cell type are stored. The *flags* provide auxiliary storage space within the cell identifier, to be used for various purposes, compare Sect. 4.

4. Adaptive grid data management. From our applications, the numerical solution of partial differential equations (PDE), we are in need to store the adaptive grid in use and the data for every cell in the grid, namely the coefficients of the solution of the PDE. This data is called the *cell data*. As explained in the last section, an adaptive grid can be represented by a set of IDs. In a typical problem involving a time–dependent PDE this set of IDs changes dynamically and a memory–efficient

storage technique to handle such a situation and to guarantee quick access to the cell data is needed. This can typically be achieved by employing a hash table. We will see that it is quite advantageous to store an $\mathbb{ID}$ within a bitstring, where the bitstring is an unsigned integer. This is memory-efficient and allows us to perform the calculation of the neighbor $\tilde{\mathbb{ID}}$ in constant time using bit manipulations on the unsigned integer. Furthermore, we will see how the associated unsigned integer can directly be used as the argument of the hash function.

4.1. ID bitstrings. In order to store the $\mathbb{ID} \equiv (path, level, basecell, block, type, flags)$, we use an unsigned integer of appropriate length (e.g. 32 or 64 bit), denoted by $\mathbb{ID}_2$, where the subscript 2 indicates the use of the binary system. Our implementation allows to partition the bits of the integer into groups of different lengths and to access the groups separately. For example

$$\mathbb{ID}_2 \equiv (\underbrace{path}_{16\text{ bits}}, \underbrace{level}_{3\text{ bits}}, \underbrace{basecell}_{8\text{ bits}}, \underbrace{block}_{2\text{ bits}}, \underbrace{type}_{1\text{ bit}}, \underbrace{flags}_{2\text{ bits}})_2 ,$$

i.e. for the *path* we reserve a group of 16 bits, for the *level* a group of 3 bits, for the *basecell* a group of 8 bits, for the *block* a group of 2 bits, for the *type* only 1 bit and the remaining group of 2 bits are used as *flags* for various purposes, e.g. in the assembling routine discussed in Sect. 4.5.2. In general we denote by $\#P$ the number of bits reserved for the path. Furthermore, there is a grouping within the path, namely if $\#B$ bits are needed to represent the child numbers $0, 1, ..., \#C-1$, then each digit in the path requires $\#B$ bits. Thus, if $\#L$ is the largest number with $\#P \geq \#L \cdot \#B$, then all cells of the grid hierarchy occurring between levels 1 and $\#L$ can be represented by the path. Accordingly we have to choose the number of bits for the level large enough, so that all numbers between 1 and $\#L$ can be represented. In the present example we have chosen an integer of 32 bits, where the grid can possess cells from level 1 to $8 = \#L$, i.e. if each cell has 4 children (i.e. for our cell types in two spatial dimensions, triangles or quadrilaterals), which can be encoded into $\#B = 2$ bits, then all cells occurring between level 1 and 8 can be represented since we have chosen $\#P = 16 = \#L \cdot \#B$. The 8 bits for the basecell allow to have a coarse grid of 256 basecells, which can be divided into 4 blocks and 2 different cell types may occur in the 4 blocks.

This technique to store the $\mathbb{ID}$ provides several advantages:

- It is very memory–efficient. The $\mathbb{ID}$ of a cell is encoded into 32 or 64 bits, which is of the same memory size as a single pointer would need on a modern computer. Note that by means of our algorithms presented in Sect. 2 and 3 we can infer the full hierarchical and spatial connectivity from $\mathbb{ID}_2$. I.e. the savings in memory over data structures that realize connectivities via pointers, is immediately visible, see Sect. 4.4.5 for details.
- It can directly be used to construct the hash function, see Sect. 4.4.1.
- It provides a cheap and simple means to perform operations only on a chosen subset of cells that are present in the adaptive grid, see Sect. 4.4.4.
- It is in particular independent of the cell type and the spatial dimension and thus enables a development of the data management algorithm independent of geometrical properties of the grid.
- The path can also be made use of in a parallelization technique based on a space-filling curve of the grid. Here the position of a cell along a space-filling curve is determined from the path of the cell using a finite state machine [15, 22]. This has been applied for logically Cartesian grids in [30] and [8]. One of our targets is to generalize this in the future to triangular, tetrahedral and hybrid grids.
- For multiphase applications it is very convenient to set up a hash map for each physical phase, i.e. phase-wise loops can easily be performed using the corresponding hash map and the particular cell data associated with each phase is also associated with the corresponding hash map, compare Sect. 4.4.

4.2. Finding neighbors in constant time using bitwise operations. We can slightly reorganize Algorithm 1 and divide it into the steps (A), (B), (C), below. The loop in the lines 4–7 is replaced by steps (A) and (B), whereas the loops in the lines 10–15 are replaced by step (C):

(A) For the given f, let $I : \{0,1,...,\#C-1\} \to \{0,1\}$ be defined by $I(c) = 1$ if we are at an insert face, i.e. if $\tilde{f}(f,c) \geq 0$, and $I(c) = 0$ if we are at a divide face, i.e. if $\tilde{f}(f,c) = -1$. Set $I_i = I(c_i)$ for *all* digits in the path.

(B) Set $I_i = 0$ for all $i > l$. Then find the largest index i for which $I_i = 1$. Set $\tilde{f} = \tilde{f}(f, c_i)$.

(C) Then set $\tilde{c}_j = \tilde{c}(\tilde{f}, f, c_j)$ for $1 \leq j \leq \#L$. Here for fixed $\tilde{f}, f$ we use the map $\tilde{c}(\tilde{f}, f, \cdot) : \{0,1,...,\#C-1\} \to \{0,1,...,\#C-1\}$. Overwrite $\tilde{c}_{i-1}, ..., \tilde{c}_1$ by $c_{i-1}, ..., c_1$.

Now $I(\cdot)$ and $\tilde{c}(\tilde{f}, f, \cdot)$ can be considered as Boolean mappings $\{0,1\}^{\#B} \to \{0,1\}^{\#B}$. Such Boolean mappings can always be written as the combination of logic operations, see [22, 28]. Furthermore, near-minimal logic operations using only few bit-manipulations can be constructed by employing Karnaugh maps, as is also detailed in [22, 28]. Finally, these logic operations can be turned into bitwise operators which act on the entire path and map from $\{0,1\}^{\#P}$ to $\{0,1\}^{\#P}$. These bitwise operators are constructed from elementary bit manipulations like `AND`, `XOR` and bit shift operations (e.g. right shift `RSH`), combined with appropriately constructed bit masks, see [21]. In essence we achieve that the Boolean mappings $I(\cdot)$ or $\tilde{c}(\tilde{f}, f, \cdot)$ are applied to all digits of the path at the same time, which can speed-up Algorithm 1. In contrast, in lines 4–7 or lines 10–15 of Algorithm 1 sequential evaluations are performed, which in the worst case require that the respective loops have to be gone through $\#L$ times. In a similar way bitwise operators were used in [24]. Finally, a fast means to realize the search for the largest index i in (B) is required. Again, using a loop to check the I_j results in $\#L$ sequential checks in the worst case. Here we use a processor instruction known as *bit scan reverse*, or shortly `bsr` to perform this task on $(0,...,0,I_l,...,I_1)_2$. This instruction is available on most modern processors, e.g. on the compatible successors of Intel 80386 [27].

Apart from the choice $\#B = 2$, Fig. 4.1 displays the general realization of Algorithm 1 using bitwise operators. The arrows drawn with continuous lines denote Boolean maps acting on a full path. Those Boolean maps being used in general are explicitly given in the figure (`AND levelmask`, `AND levelinsertmask`, `AND insertmask`, `OR`), the remaining ones depend on the cell type and represent the maps I and the table $\tilde{c}$.

We discuss Fig. 4.1 in more detail for the cell type triangle and the case that the neighbor cell across face $f = 0$ is sought. Table $\tilde{f}(0,\cdot)$ of Fig. 2.7 reveals that $I(c) = 1$ iff the higher of the two bits needed to represent c is zero. Thus we calculate (`NOT` $(c_{\#L},\ldots,c_1)_2$) `AND oddmask`, where `oddmask` $= (\ldots,1,0,1,0,1,0)$ is a bitstring of length $2\,\#L$ with every odd bit set. We shift the result by one bit to the right to obtain $(I_{\#L},\ldots,I_1)_2$. Before applying `bsr` we have to remove misleading information, i.e. $(I_{\#L},\ldots,I_1)_2$ `AND levelmask` $= (0,\ldots,0,I_l,\ldots,I_1)_2$, where only the bits corresponding to $c_l,\ldots,c_1$ are set in `levelmask` and l is the level of the cell. The `levelmask`, and all other masks, are tabulated. Now `bsr` locates the insert face at level $i =$ `bsr`$((0,\ldots,I_l,\ldots,I_1)_2)\,\mathrm{div}\,\#B$, where div stands for integer division. If $i = -1$, then the bitstring only has zero bits, i.e. face $f = 0$ is part of the base cell boundary. Since then there is no neighboring cell, the algorithm terminates. Otherwise $\tilde{f} = \tilde{f}(0, c_i)$ can be obtained from the corresponding digit c_i. Next a bitwise version of $\tilde{c}(\tilde{f}, 0, c)$ is applied to the original path. The valid fine level digits in the result are extracted using a bitwise `AND` with the `levelinsertmask`, which has bits set only where digits of level $i,\ldots,l$ are located. In order to copy the coarse level digits, the `insertmask`, which only has bits set at digits corresponding to levels coarser than the insert face, is applied on the original path using a bitwise `AND` operation. Finally the two results obtained sofar in the left and in the right column of the figure are combined with a bitwise `OR` operation (dashed arrow on bottom of Fig. 4.1) to obtain the path of the neighbor cell.

For each $f, \tilde{f}$ the corresponding bit operations, made up of logic operations and masks, that realize $I(\cdot)$ and $\tilde{c}(\tilde{f}, f, \cdot)$ are implemented and called when needed within the steps (A), (B), (C). In the worst case steps (A), (B), (C) together require 8 bitwise operations for the cell type triangle and at most 46 bitwise operations for the cell type tetrahedron. Since each bitwise operation and `bsr` are performed within very few processor clock cycles, the spacial connectivity across grid faces is established very fast.

In order to put the work load for neighbor finding into relation to other tasks within a PDE solver, let us look at the face term $\int_F a(x,t)\phi(x)u(x)dx$ that arises in the DG formulation in a simple

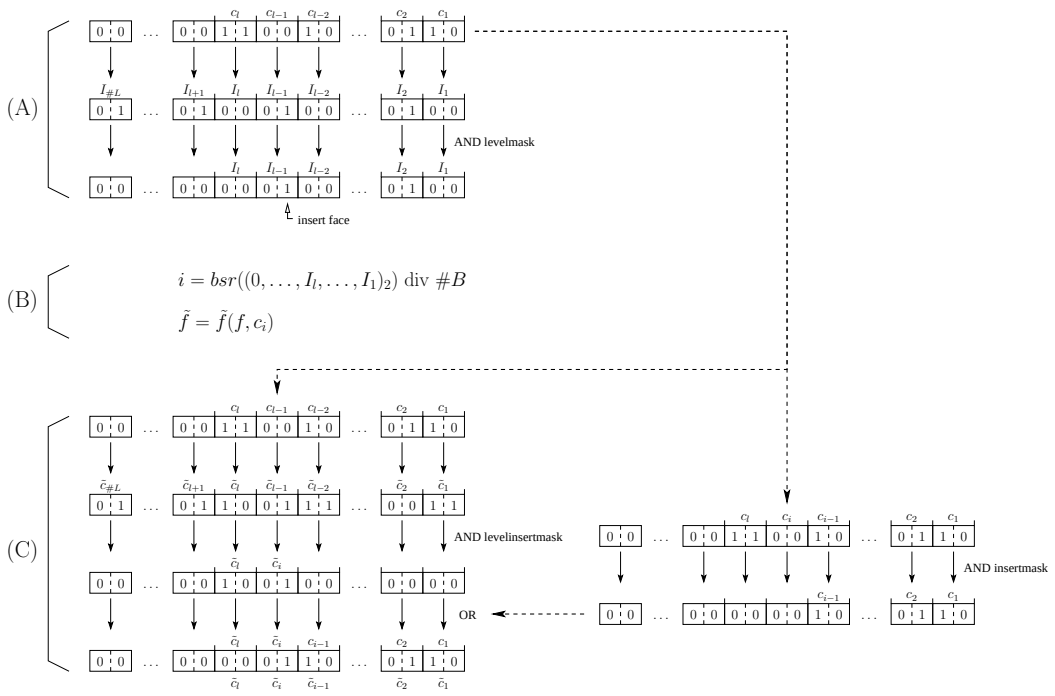


Fig. 4.1: Neighbor finding in constant time for 2D triangular paths at face $f = 0$ using two-bit digits.

transport problem. We consider two dimensional grids, so that the integral is a line integral, and we assume that the normal transport velocity a is constant, so that a Gaussian quadrature rule using m points has order $2m - 1$ of exactness. If we assume the polynomial ansatz and test functions u and ϕ to be of order p, we need at least a quadrature rule with $p + 1$ points to exactly integrate the product. So not neglecting the quadrature weight, we need at least 2 multiplications per quadrature point. Overall we have $2(p + 1)$ multiplications per edge to calculate the above edge integral. Note that this integral has do be evaluated for each of the $(p+1) \cdot (p+2)/2$ test functions associated with each of the two cells adjacent to face F. Due to the fact that on modern processors the workload for one double multiplication and one bitwise operation are roughly the same, the workload for the face quadrature easily dominates the 8 bitwise operations required for finding the neighbor.

Neighbor finding across base cell boundaries. If `bsr` delivers the insert level $i = 0$, i.e. if above no neighboring cell was found within the base cell, then Algorithm 2 reduces to the application of the table $\tilde{c}$ that is stored on the base cell for each face. As in the search for a neighboring cell within the base cell, this mapping can be interpreted as a Boolean mapping on the full path and can be executed in constant time by few bitwise operations. The mapping can be applied directly to the whole path, because it is known that no insert face was found. As there are only finitely many different tables $\tilde{c}$, it suffices to prepare the bit operations for each table and store the corresponding table number for each face on the base cell.

4.3. Leaf-, ante- and basecells. The cells that constitute the coarse grid have already been introduced as the *base cells* in Sect. 3, and since the base cells are fixed, we can store the base cell data and the coarse grid connectivity in an array which we denote by BA. The domain covered by the coarse grid is locally refined to the grid on which we have chosen to discretize a PDE. Then we call the cells in this grid the *leaf cells*. We adopt the convention, that we never let a base cell become a leaf cell, i.e. base cells are considered as cells on level 0, whereas all leaf cells are at least of level 1. This is very convenient, since different data are stored on base and leaf cells. The cells that can be

found in the hierarchy between base cells and leaf cells are the *ante cells*. I.e. any ancestor of a leaf cell, which is not a base cell, is an ante cell. Both the set of ante cells and the set of leaf cells are each stored within a hash table, compare Sect. 4.4. These two hash tables are denote by LHT (Leaf cell Hash Table) and AHT (Ante cell Hash Table).

4.4. Hash table for cell data. The pair of cell identifier and associated cell data, $(\mathbb{ID}_2, \textit{cell data})$, present for each cell in an adaptive grid, is the essential data we have to deal with. Adding and deleting cells could be handled very efficiently and with low memory cost by a linked list of these pairs. But if we wanted to know if a cell defined by a given $\mathbb{ID}$ exists in the list or we wanted to access the associated cell data, then a time–consuming linear search would be needed. On the other hand, if we ensured fast access to the cell data by storing it in an array at an array index uniquely associated with $\mathbb{ID}$, then we would waste an enormous amount of storage, since we had to provide an array with as many entries as potentially attainable $\mathbb{ID}$s. A beneficial compromise between these two rather naive points of view is a *hash table*. It is a well–known data structure, which achieves dense storage of data items via a *hash function*, while at the same time access of the data, respectively the check for the availability of the data, is performed in constant time, i.e. with expected complexity $\mathcal{O}(1)$. See [12, chapter 12] and [20, section 6.4] for an introduction to hash tables.

In the general setting of hash tables, the pair $(\mathbb{ID}, \textit{cell data})$ is denoted as $(key, value)$. The set of all potentially attainable *keys* is called the *universe of keys*. For the fast access property of a hash table, a high quality hash function, and an efficient *collision handling*, see Sect. 4.4.1, have to be found. We use a hash table based on *chaining*, i.e. the pairs $(key, value)$ of interest are stored within an array of p linked lists, see Fig. 4.2. The array entries are denoted as *slots* or *buckets*. A few other types of hash tables exist [12]. A hash function is a mapping from the universe of possible keys onto the slots, i.e. onto the set $\{0, 1, 2, ..., p-1\}$.

To further improve the performance of the hash table, we set up the *memory heap*, a special memory management for the hash table, see Sect. 4.4.2. Following the concepts of the C++ standard library [18], we provide iterators to loop over the data items stored in the hash table. A special feature of the implementation of our hash table is that all data items corresponding to the same refinement level are connected by a doubly–linked list, see Sect. 4.4.3. This level–link can be seen in Fig. 4.2 and is a valuable device when employing a multilevel strategy in the PDE–solution method.

4.4.1. Hash function and collision handling. The hash function h maps the universe of keys onto the numbers $\{0, 1, 2, ..., p-1\}$, where p is the number of slots available,

$$\begin{array}{rlll} h: & \textit{universe of keys} & \to & \{0, 1, 2, ..., p-1\}, \\ & k & \mapsto & s = h(k). \end{array}$$

The evaluation of the hash function is required to be deterministic and of constant complexity. If we now would like to find the cell data v corresponding to a given key k, then we know that if the pair (k, v) is available, then it can be found in the linked list attached to slot $s = h(k)$.

In our case, ignoring the $flag$–bits, the remaining bits of $\mathbb{ID}_2 \equiv (path, level, basecell, block, type, 00...0)_2$ form a unique identifier of a cell. We set the $flag$–bits to 0 for evaluating the hash function and apply a modulus operation on the integer $(path, level, basecell, block, type, 00...0)_2$:

$$\begin{array}{rlll} h: & \textit{universe of keys} & \to & \{0, 1, 2, ..., p-1\}, \\ & \mathbb{ID}_2 & \mapsto & h(\mathbb{ID}_2) = (path, level, basecell, block, type, 00...0)_2 \bmod p. \end{array}$$

As mentioned before, if $s = h(k)$ is known, a further search is needed within the list associated with slot s in order to find the pair (k, v) in the list.

The hash function cannot be injective, since the hash table makes only sense, if the cardinality of the universe of keys is much bigger than the number of slots available. A *collision* occurs, if h maps two keys to the same slot. Our choice to associate a linked list (called a *chain* in this context) with each slot, resolves this problem: If k and $s = h(k)$ are given, a linear search within the chain associated with slot s is performed in order to find k in the list. Besides other approaches, chaining is a classical way to resolve collisions [20].

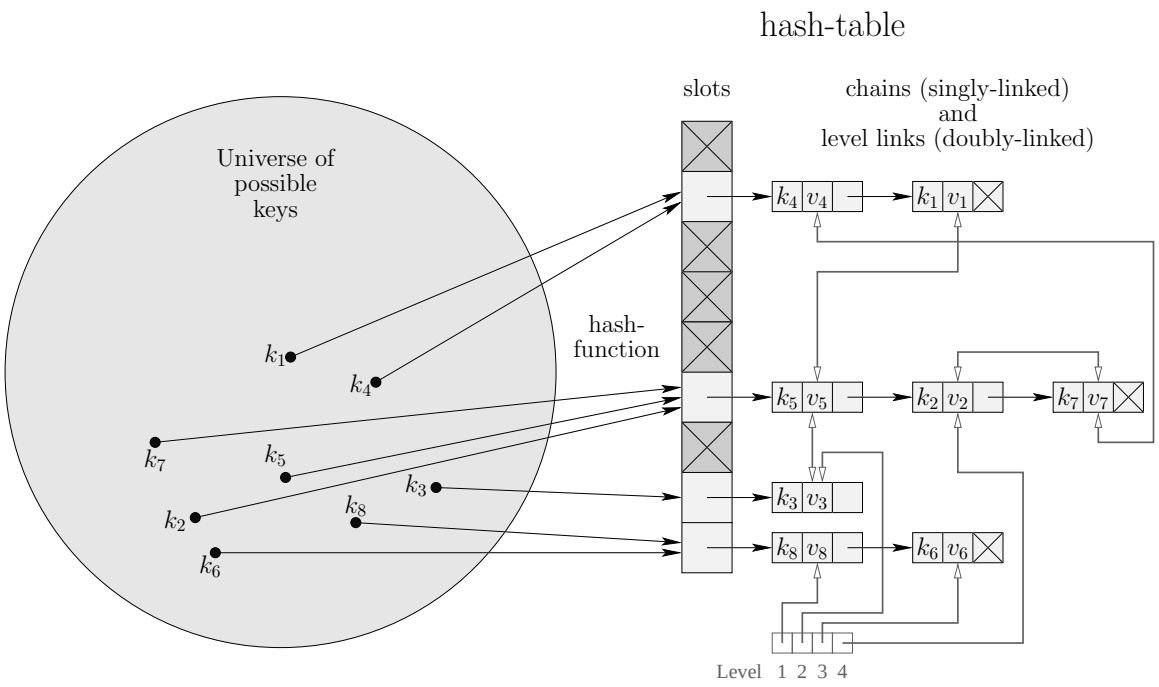


Fig. 4.2: Hash table: Pairs $(key, value) \equiv (\mathbb{ID}_2, cell\,data)$ are shortly denoted by (k_j, v_j) for $j \in \mathbb{N}$.

In order to guarantee efficiency, all chains should be as short as possible and of almost the same length to minimize the costs of linear search, which is of complexity $\mathcal{O}(M)$ when M is the length of the chain. Furthermore to reduce the number of collisions, the number of slots p should not be a power of 2, because if $p = 2^m$, then $h(k)$ only depends on the last m bits of the k. Best distributions can be expected, if p is a prime number, and not too close to a power of 2, see [12].

4.4.2. Memory–heap. When the grid changes dynamically, excessive allocation and deletion of single $(key, value)$-pairs in the hash table can be quite time-consuming. To overcome this problem, we set up a memory management framework for the hash table that is called memory–heap and allocates or deallocates memory block-wise. Starting from an estimation of the storage space needed, a first block of memory is allocated. If more data needs to be stored, more smaller blocks are allocated on demand.

4.4.3. Iterators and level-wise connection. We can traverse all leaf cells by looping through the slots of LHT and running at each slot through the attached chain. We provide an *iterator* for this loop over all leaf cells. In applications it can be advantageous to run through the cells in the order of the levels. For this purpose we have placed pointers in the hash table, which connect cells of the same level. These pointers constitute a doubly linked list for each level, see Fig. 4.2. Whenever a cell is deleted from the grid or new ones are created, these lists have to be updated.

All applications discussed in Sect. 4.5 make use of this level-wise links, in particular multilevel methods for the solution of PDEs, see Sect. 4.5.4, make excessive use of these links. Here typically the algorithms not only loop through the leaf cells, but also through all ancestors of the leaf cells, i.e. through the corresponding hash table AHT, for which level-wise links and iterators are provided as well.

4.4.4. IDsets. Some operations are collective on a set of cells, e.g. a set of leaf cells may be proposed for refinement and we want to replace each of these leaf cells in LHT by its children. Again

it may be very useful, compare Sect. 4.5.3, to loop level-wise through the set of leaf cells. For such a situation we provide hash tables without value type, which we call $\mathbb{ID}$*sets*. I.e. iterators and level-wise connection can be used for $\mathbb{ID}$sets as described in Sect. 4.4.3.

4.4.5. A comparison to pointered trees. In classical codes [3, 26], hierarchical as well as spatial connectivity of the cells are represented by pointer structures, which can be quite memory-consuming. As an example, let us consider the refinement rules for triangles and tetrahedra given in Sect. 2. Then each triangle needs 1 pointer to refer to the parent cell, 4 pointers to refer to the child cells and, using a simply graded grid, 6 pointers to the neighbor cells, that is a total of 11 pointers per cell. For a tetrahedral cell one needs to store 1 pointer to the parent cell, 8 pointers to children and 16 pointers to neighboring cells, so overall 25 pointers. All these pointers have to be kept up-to-date, e.g. when refining a cell, any connectivity related to that cell and represented by a pointer has to be changed.

On modern 64 bit computers, a pointer requires 64 bits to be stored, which is exactly the same as for a double-precision floating-point number. Assume we approximate each unknown of our solution with piecewise quadratic polynomials on a triangulation. Then we have 6 degrees of freedom, i.e. 6 doubles, per triangle and unknown, which is roughly half the amount of storage required for the pointers that represent the connectivity between triangles.

Using our approach based on cell identifiers, there are no pointers attached to the cells for establishing connectivity. Instead, we store for each cell one $\mathbb{ID}$ as an unsigned integer, which is of a size comparable to that of a pointer. Remember that our storage of the grid is not totally free of pointers, but requires only very few of them. For the hash map, we use one pointer per slot in the hash table and additionally one pointer per cell for chaining. Note that the number of pointers used per cell is also independent of the dimension and type of the cells. Furthermore it is a very useful feature to have the possibility to loop level-wise over the cells. So, as many classical pointer-based codes also do [3, 26], we invest two more pointers per cell for a doubly-linked list that connects all cells within the same level. We chose to use a doubly-linked list to be able to remove a cell from the list in constant time.

Comparing the computational work of the two grid management approaches, the principle differences are the following: Our approach calculates spatial connectivity, whereas cell neighbors are accessible via pointers in the tree-based approach. As detailed in Sect. 2.7 and 3.3, the full information needed for matching face quadrature points in DG assembling routines is also provided when we calculate the spatial connectivity, without requiring any additional storage or calculations. Furthermore, when refining or coarsening the grid, in the tree-based approach all hierarchical and spatial connectivities have to be updated by rearranging pointers, which is fully omitted in our approach.

4.5. Applications. Next we outline how typical algorithmic tasks that arise in the implementation of DG methods can be performed with our data structure.

4.5.1. Ensuring simply graded grids. Let us show how to ensure that a grid is simply graded, compare Sect. 3.3. Given an arbitrarily refined grid, which is a partition of the coarse triangulation, we would like to refine this grid, such that a simply graded grid results with as few as possible cells.

In Sect. 4.4.3, we saw that we can loop level-wise through our hash map. Using this feature, we now run through all leaf cells of a grid, which are stored in the leaf cell hash table (LHT), from the finest to the coarsest level. for each leaf cell we determine the parent cell, loop over the faces of the parent cell and check if the face is either part of the boundary or if the face neighbor of the parent cell is present in the grid. If we are not at the boundary and the face neighbor is not present, then there is at least an ante cell at level 1 which we can refine until the missing face neighbor exists. This approach is also expressed in Algorithm 3, where we rely on the fact that an ante cell hash table (AHT) is present, as explained in Sect. 4.3.

For this algorithm, the level-wise operation from finest to coarsest level is crucial, because while traversing cells of a level, there are only cells of coarser levels inserted. So the linked list of cells is not altered at the level that is currently processed. Note that testing if $\mathbb{ID} \in LHT$ requires a search for the key $\mathbb{ID}$ in the hash table LHT.

As an example, starting from a coarse grid we loop several times through the grid and refine in each loop those cells that are cut by a sphere. We apply Algorithm 3 to this grid, which is not yet graded. A simply graded grid results which is shown in Fig. 4.3 for a 2D and a 3D hybrid case.

Algorithm 3 Grading algorithm: Ensure that adjacent cells differ in level at most by 1.

```
1: for (l = #L; l ≥ 1; l − −) do                                  ▷ loop over levels from fine to coarse
2:     for (iter = LHT.begin(l); iter! = LHT.end(l); + + iter) do  ▷ loop over (leaf)cells of level l
3:         get cell data, including 𝕀𝔻, from LHT entry that iter points to;
4:         for (face f = 0, ..., #F − 1) do
5:             use Algorithm 2 to determine 𝕀̃𝔻, the neighbor of 𝕀𝔻 across face f;
6:             if (𝕀̃𝔻 ∉ AHT) then                                ▷ ensure that neighbor cell is not finer
7:                 𝕀̃𝔻_a = 𝕀̃𝔻;
8:                 while (𝕀̃𝔻_a ∉ LHT) do                          ▷ find ancestor leaf cell
9:                     𝕀̃𝔻_a = parent(𝕀̃𝔻_a);
10:                end while
11:                for (j = level(𝕀̃𝔻_a); j < level(𝕀̃𝔻) − 1; j + +) do
12:                    determine 𝕀̃𝔻_a of ancestor cell of 𝕀̃𝔻 at level j;
13:                    refine 𝕀̃𝔻_a;                              ▷ refine until parent of 𝕀̃𝔻 is present
14:                end for
15:            end if
16:        end for
17:    end for
18: end for
```

4.5.2. Assembling Loop for the Discontinuous Galerkin Method. We assume a PDE to be given that is spatially discretized using a DG method. Either when setting up the linear system of discrete equations (in case we have a stationary and linear problem, see [1]) or when advancing the equations in time (in case we apply an explicit time integrator for a time dependent problem, see [10]), then we have to evaluate the volume terms and the face terms appearing in the DG discretization. In order to assemble these terms we have to loop through the grid with the help of our data structure. This assembling loop is given in Algorithm 4. Here we assume that the leafcells constitute a simply graded grid.

In this loop we make use of the flags introduced in Sect. 4.1 to mark cells that have already been processed. Thus in line 1 we first set all these flags to $false$. In order to (efficiently) handle hanging vertices, we loop through the leafcells level-wise, going from fine to coarse levels (lines 2-3), making use of the level links within the hash table. Reaching an entry of the hash table we have access to the corresponding cell data and $\mathbb{ID}$ of the cell. The cell data can be used to evaluate volume terms, whereas $\mathbb{ID}$ can be used to find $\tilde{\mathbb{ID}}$, the identifier of the neighbor cell (line 7). Typically the face terms establish the coupling of neighboring cells within a DG discretization, so that the cell data corresponding to $\mathbb{ID}$ and $\tilde{\mathbb{ID}}$ are needed when evaluating the face terms. In order to have access to the cell data of $\tilde{\mathbb{ID}}$ and the corresponding flags we have to find $\tilde{\mathbb{ID}}$ in the hash table, which is done in line 9, in case we are not at a boundary face. Note here, that the query "**if** ($\tilde{\mathbb{ID}} \in LHT$)" is accomplished by searching for $\tilde{\mathbb{ID}}$ in the leaf cell hash table LHT. Note also that the flags of $\tilde{\mathbb{ID}}$ found by the neighboring algorithm in line 7 are meaningless. We first have to find $\tilde{\mathbb{ID}}$ in the hash table, where $\tilde{\mathbb{ID}}$ is stored with the flags inherent to the corresponding leafcell. If the flag of $\tilde{\mathbb{ID}}$ has not yet been processed, then the terms for the current face still have to be assembled. If we cannot find $\tilde{\mathbb{ID}}$ in the hash table, i.e. if we are led to line 14, then hanging vertices occur at the current face. Since the leafcells form a simply graded grid, the neighbor of $\mathbb{ID}$ in the grid is either of one level higher or of one level lower. If it is of one level higher, then the face has already been processed before. Thus it remains to check if the parent cell of $\tilde{\mathbb{ID}}$ can be found in the hash table (line 16). Note that in this case, the current face is a full face for $\mathbb{ID}$ but only part of a face for $\tilde{\mathbb{ID}}_p$, the parent of $\tilde{\mathbb{ID}}$. In any case

that may occur, we have already found all information to evaluate the face terms via the quadrature rule (2.5) or (3.2). These evaluations occur in line 12 if $\tilde{\mathbb{D}}$ is the neighbor of $\mathbb{D}$ in the grid, in line 18 if $\tilde{\mathbb{D}}_p$ is the neighbor of $\mathbb{D}$ in the grid, and in line 22 if the face is on the boundary. Note that applying (3.2) in line 22 requires first to evaluate table $e(\cdot,\cdot)$ to obtain $\tilde{e} = e(\tilde{f}, \tilde{c}_l)$, compare Sect. 3.3.

Algorithm 4 Assembling loop for the Discontinuous Galerkin Method on simply graded grid

```
1: set assembleflag to false in 𝔻 for all leafcells in LHT;
2: for (l = #L; l ≥ 1; l − −) do                                  ▷ loop over levels from fine to coarse
3:     for (iter = LHT.begin(l); iter! = LHT.end(l); iter + +) do   ▷ loop over (leaf)cells of level l
4:         get cell data, including 𝔻, from LHT entry that iter points to;
5:         assemble volume terms;
6:         for (face f = 0, . . . , #F − 1) do
7:             use Algorithm 2 to determine f̃, o and 𝔻̃ for 𝔻 across face f;
8:             if (f̃ ≠ −1) then                                   ▷ neighbor found at inner face
9:                 if (𝔻̃ ∈ LHT) then                               ▷ neighbor is on same level
10:                    if (not 𝔻̃.assembleflag) then               ▷ neighbor cell has not been processed
11:                        get corresponding cell data for 𝔻̃ from LHT;
12:                        assemble face terms;                    ▷ use (2.5)
13:                    end if
14:                else                                           ▷ hanging vertices
15:                    determine 𝔻̃_p, the parent cell of 𝔻̃;
16:                    if (𝔻̃_p ∈ LHT) then                        ▷ 𝔻̃_p has hanging vertices on face f̃
17:                        get cell data for 𝔻̃_p from LHT;
18:                        assemble face terms;                    ▷ use (3.2)
19:                    end if
20:                end if
21:            else                                               ▷ boundary face
22:                assemble face terms;                           ▷ use (2.5) with ṽ from boundary data
23:            end if
24:        end for
25:        set 𝔻.assembleflag;        ▷ indicating that the cell and all its faces have been processed
26:    end for
27: end for
```

4.5.3. Adaptation of simply graded grids. In practical codes, grid adaptation is required to improve the quality of solutions. Assuming a simply graded grid is given, we can formulate an algorithm that keeps the grid simply graded during adaptation as follows:

Assume an adaptation criterion is given that decides if a leafcell should be kept in the grid, refined or coarsened. We collect all cell identifiers proposed for refinement in `refineset` and those proposed for coarsening in `coarsenset`. Note that the two sets are in fact hash tables that have the same structure as LHT, i.e. they can be processed level-wise. Note also that since the input grid is simply graded, for any cell the adjacent cells differ in level by -1, 0 or $+1$. Consequently, after the cell is refined, the level difference is -2, -1 or 0. Only in the first case the grading condition is not fulfilled, which is resolved by adding the neighbor cell to `refineset`. As the `refineset` is processed level-wise and the neighbor cell is on a coarser level, it is processed later on. Similarly, if for any cell proposed for coarsening an adjacent cell differs in level by $+1$, then coarsening is prohibited. Here we implicitly assume that locally finer grids produce more accurate solutions.

An algorithm that updates the grid by combining refinement, coarsening and grading, consists of the following chronological steps. A detailed formulation of the algorithm can be obtained along the lines of Algorithm 4, including details like boundary treatment, which will not be covered here.

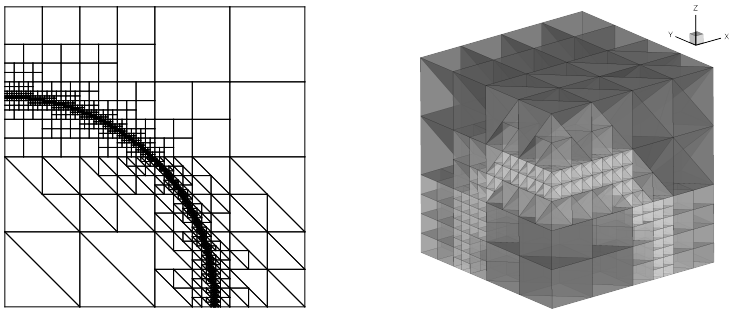
- For refinement, we loop through `refineset` from fine to coarse levels. For each cell to be

refined, we generate the neighbors across all faces according to Algorithm 2. We determine the parent cell of each neighbor. If a parent cell is a leaf cell, then it is inserted into `refineset`. Furthermore we remove it from `coarsenset`, if present, as it will no longer be a leaf cell. Note that the grid is simply graded after completing refinement.

- A cell should be coarsened only, if all its siblings are also marked for coarsening. Thus we fill a hash table called `CHT` with the parent cells of the cells in `coarsenset`. Here the key is the identifier of the parent cell and the associated value is an integer that counts how many of its children are in `coarsenset`.
- To coarsen the grid, we traverse through `CHT` from fine to coarse levels. For each cell in `CHT` with counter value $= \#C$ we loop over the faces and determine the neighbor and $\tilde{f}$ according to Algorithm 2. If the neighbor is a leaf cell, we proceed to the next face. Otherwise we determine the children of the neighbor which are adjacent to the cell from `CHT`. These children are easily identified by a lookup in a table depending on $\tilde{f}$. If any of these children is not a leaf cell, then the present cell from `CHT` will not be coarsened and we proceed to the next cell in `CHT`. Otherwise we proceed to the next face.
  If all faces have been checked, without proceeding to the next cell, the current cell is coarsened and after that we continue with the next cell from `CHT`.

Note that the coarsening loop like the one for refinement is performed from fine to coarse levels, because coarsening of a cell only takes place, if the levels of adjacent cells are appropriate. Unlike refinement coarsening does not force the cells in its surrounding to change its level. Assume all cells with level above 1 of a simply graded adaptive grid are marked for coarsening and we expect every cell to be coarsened. If the algorithm started with coarse levels, hanging nodes would possibly prohibit the coarsening of coarse cells, even if the adjacent cells are coarsened lateron.

We slightly modify the example from Sect. 4.5.1: Now, starting from a coarse grid we loop again several times through the grid, insert each cell that is cut by a sphere into `refineset` and all others into `coarsenset`. In each loop we apply the algorithm described above. Again the grids shown in Fig. 4.3 emerge.



(a) 2D grid (b) 3D grid

Fig. 4.3: Simply graded hybrid adaptive grid in 2 and 3 spatial dimensions.

4.5.4. Multilevel data and fast wavelet transform. Let us associate with each cell K of a given cell type a basis of n functions $\phi_1^K, \phi_2^K, \ldots, \phi_n^K$ with support on K, which typically are given by a set of Finite Element shape functions, extended by 0 outside of K. Then we can represent functions of the form

$$u \;=\; \sum_{K \in LHT} \; \sum_{i=1}^{n} c_i^K \, \phi_i^K \tag{4.1}$$

on the leaf cell grid, and for each leaf cell K we have

$$u|_K \;=\; \sum_{i=1}^{n} c_i^K \phi_i^K.$$

The array of coefficients $(c_1^K, c_2^K, \ldots, c_n^K)$ represents the cell data for u on K. Assume that K is of level l and that its children $K'_0, K'_1, \ldots, K'_{\#C-1}$ are of level $l+1$. Typically, then we have

$$\mathrm{span}\{\phi_1^K, \phi_2^K, \ldots, \phi_n^K\} \subset \mathrm{span}\{\phi_1^{K'_0}, \phi_2^{K'_0}, \ldots, \phi_n^{K'_0}, \phi_1^{K'_1}, \ldots\ldots, \phi_n^{K'_{\#C-1}}\}$$

and one can construct a so-called wavelet basis $\psi_1^K, \psi_2^K, \ldots, \psi_{n\cdot(\#C-1)}^K$ with support on K and such that

$$\mathrm{span}\{\phi_1^K, \phi_2^K, \ldots, \phi_n^K, \psi_1^K, \psi_2^K, \ldots, \psi_{n\cdot(\#C-1)}^K\} = \mathrm{span}\{\phi_1^{K'_0}, \phi_2^{K'_0}, \ldots, \phi_n^{K'_0}, \phi_1^{K'_1}, \ldots\ldots, \phi_n^{K'_{\#C-1}}\}.$$

For particular constructions of wavelet bases, see for example [19].

Assume now that $K'_0, K'_1, \ldots, K'_{\#C-1}$ are leaf cells and that v is expanded into the corresponding basis functions, then on $K = K'_0 \cup K'_1 \cup \cdots \cup K'_{\#C-1}$ we can represent $v|_K$ in the *single scale form*

$$v|_K \;=\; \sum_{j=0}^{\#C-1} \sum_{i=1}^{n} c_i^{K'_j} \phi_i^{K'_j}, \tag{4.2}$$

and also in the *two scale form*

$$v|_K \;=\; \sum_{i=1}^{n} c_i^K \phi_i^K \;+\; \sum_{i=1}^{n\cdot(\#C-1)} d_i^K \psi_i^K. \tag{4.3}$$

Transforming between (4.2) and (4.3) is a change of basis, where in (4.3) the coarse level features $\sum_{i=1}^{n} c_i^K \phi_i^K$ have been separated from the fine level features $\sum_{i=1}^{n\cdot(\#C-1)} d_i^K \psi_i^K$. The coarse level features are also denoted as the *mean* and the fine level features as the *detail*.

Assume now that our leaf cell grid is a uniformly refined grid of level $\#L$, and that a function u is given by (4.1). Then (4.1) is a single scale representation and we collect all coefficients of (4.1) in the array $C^{\#L}$. The corresponding grid of parent cells is given by all the ante cells of level $\#L-1$. We consider any ante cell K of level $\#L-1$ and its children $K'_0, K'_1, \ldots, K'_{\#C-1}$, which are leaf cells of level $\#L$. Then we can write $u|_K$ in the single scale representation of the form (4.2). This single scale representation of level $\#L$ can be transformed into the two scale form (4.3) with mean on the scale of level $\#L-1$ and details on the scale of level $\#L$. We collect all mean coefficients c_i^K occurring for all ante cells of level $\#L-1$ in the array $C^{\#L-1}$ and similarly all detail coefficients d_i^K in the array $D^{\#L-1}$, although we note again that the latter ones correspond to information on the scale of level $\#L$. This separation into means and details can be applied recursively, to transform next the means represented by $C^{\#L-1}$ into means $C^{\#L-2}$ and details $D^{\#L-2}$. Here $C^{\#L-2}$ contains the means on the ante cells of level $\#L-2$. This process can be continued until we reach the base cells with means C^0 and details D^0. It is schematically depicted in Fig. 4.4 and known as the *pyramid algorithm*. Note that at the end of the process the coefficient array $C^{\#L}$ has been fully decomposed into the details $D^{\#L-1}, D^{\#L-2}, \ldots, D^0$ plus the base cell means C^0, which is known as a *wavelet or multi scale representation* of u. The process can also be reversed by determining $C^{\#L}$ from the wavelet representation.

The wavelet representation of u is conveniently stored by distributing the coefficients appearing in D^l to the ante cells of level l they belong to. This requires to store for each ante cell an array of $n\cdot(\#C-1)$ coefficients, which is independent of the level l. For each base cell we also have to store the corresponding mean coefficients from C^0, which results in $n\cdot\#C$ coefficients to be stored for each base cell. In contrast, the array $C^{\#L}$ that contains the single scale coefficients of u can be

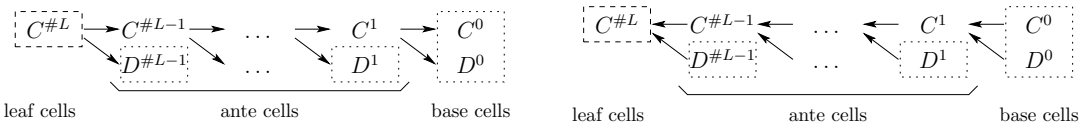


Fig. 4.4: Pyramid algorithm for fast wavelet transform and its inverse. The dashed box contains the single scale coefficients $C^{\#L}$ stored on the leaf cells and the dotted boxes mark the multiscale coefficients D^J for $0 \leq J < \#L$ stored on ante and base cells.

stored by assigning to each leaf cell K the n coefficients $(c_1^K, c_2^K, \ldots, c_n^K)$. Thus leaf cells, ante cells and base cells require different cell data to represent u. This is in fact another reason for setting up the separate data units LHT, AHT and BA to manage the grid and its hierarchy.

Note that in fact it is not required to have a uniformly refined leaf cell grid in order to perform the pyramid algorithm and that with a nonuniformly refined leaf cell grid the algorithm works analogously and produces again $n \cdot (\#C - 1)$ detail coefficients for each ante cell and $n \cdot \#C$ coefficients for each base cell.

Algorithm 5 shows how to implement the fast wavelet transform/pyramid algorithm on an adaptive grid. It proceeds from fine to coarse levels in AHT, utilizing the level links within AHT through the corresponding iterator. For each ante cell and each base cell visited a transformation from (4.2) to (4.3) is performed. For such a transformation typically a single *mask matrices* g and h can be constructed that can be applied throughout the hierarchy to perform the transformations

$$g : (c_1^{K'_0}, c_2^{K'_0}, \ldots, c_n^{K'_0}, c_1^{K'_1}, \ldots\ldots, c_n^{K'_{\#C-1}}) \mapsto (c_1^K, c_2^K, \ldots, c_n^K),$$
$$h : (c_1^{K'_0}, c_2^{K'_0}, \ldots, c_n^{K'_0}, c_1^{K'_1}, \ldots\ldots, c_n^{K'_{\#C-1}}) \mapsto (d_1^K, d_2^K, \ldots, d_{n\cdot(\#C-1)}^K).$$

Similarly, the inverse fast wavelet transform can be accomplished by using appropriate masks $\tilde{g}$ and $\tilde{h}$ and looping from coarse to fine level.

Algorithm 5 Multilevel loop: Fast wavelet transform

```
for (l = #L − 1; l ≥ 1; l − −) do                          ▷ loop over levels from fine to coarse
    for (iter = AHT.begin(l); iter! = AHT.end(l); iter + +) do   ▷ loop over ante cells of level l
        get 𝕀𝔻 for cell in AHT that iter points to;
        determine children 𝕀𝔻_i of 𝕀𝔻   for 0 ≤ i < #C;
        get children's cell data (coefficient vector c_i^{l+1})   from LHT or AHT for 0 ≤ i < #C;
        calculate c^l and d^l   by applying masks g and h on (c_i^{l+1})_{0≤i<#C};
        store coarse and detail coefficients c^l and d^l on cell 𝕀𝔻;
    end for
end for
for (b = 0; b < #basecells; b + +) do                      ▷ loop over base cells
    get 𝕀𝔻 for b-th base cell from base cell array;
    determine children 𝕀𝔻_i of 𝕀𝔻   for 0 ≤ i < #C;
    get children's cell data (coefficient vector c_i^{l+1})   from LHT or AHT for 0 ≤ i < #C;
    calculate c^0 and d^0   by applying masks g and h on (c_i^1)_{0≤i<#C};
    store coarse and detail coefficients c^0 and d^0 on base cell 𝕀𝔻;
end for
```

5. Conclusion. We presented a concept for managing multilevel adaptive grids, which aims at supporting efficiently and with low storage requirements the numerical solution of PDEs based on DG discretizations. The concept also provides a unifying framework for handling various cell types and even hybrid grids composed of different cell types. In contrast to classical data structures that employ

pointers to store hierarchical and spatial connectivity, our data structures relies on cell identifiers that uniquely characterize grid cells, algorithms that provide hierarchical and spatial connectivity and a hash table to store the cells and the corresponding cell data. Independent of the cell type, we have developed a theory to derive the algorithm for the spatial connectivity. Information needed to apply quadrature formulas on grid faces, necessary when assembling discrete DG equations, is also provided by the algorithm. The theory formulates conditions on the refinement rule used for an arbitrary cell type under which the algorithm works. We display standard refinement rules for quadrilaterals, triangles, cuboids, tetrahedra in our framework, which fulfill these conditions. Generalizations to hybrid grids and cell types composed of the above in tensor product fashion are straightforward and examples are given. A fast realization of the spatial connectivity algorithm is given by reformulating parts of the algorithm in terms of bitwise operations on unsigned integers which represent the cell identifiers. These bitwise operations depend on the cell type. The unsigned integer used to represent the cell identifier is used as the key in the hash table in order to address cells. Since unsigned integers are used independent of the cell type, the hash table also constitutes a cell type independent part of the concept.

The concept has been successfully applied in practical implementations by the authors, and the authors have used it in conjunction with DG discretizations in other publications. Generalizations to anisotropic refinements and extensions of the concept to include parallelization and parallel multilevel preconditioners are currently under construction.

Appendix A. Supported cell types. Besides the cell types given in the previous chapters, all algorithms presented before are also applicable to quadrilateral cells in 2D and cuboid and prism cells in 3D. The Cartesian cell types are much simpler than triangles and tetrahedra. Furthermore, prisms are simply constructed as tensor-products of triangles and quadrilaterals, and thus we present only the tables for quadrilaterals, cuboids and prisms and restrict ourselves to a few comments for these cell types.

The characteristic numbers #D, #N, #F and #NF as well as #C and #CF have already been given in Fig. 2.1 and Fig. 2.3 for all cell types. Since the -1 entries in $\tilde{c}(\tilde{f}, f, c)$ indicate that the corresponding argument $(\tilde{f}, f, c)$ does not occur, we can in fact write table $\tilde{c}(\tilde{f}, f, c)$ for cell type triangle into one table, which is independent of $\tilde{f}$, compare Fig. 2.11. This coincidence also comes up for quadrilaterals and cuboids, so that we give the corresponding table in the compact form $\tilde{c}(f, c)$, see Fig. A.1 and Fig. A.2. The same applies for the cell type prism, see Fig. A.4.

Quadrilaterals. The tables for quadrilaterals are presented in Fig. A.1. The numbering of the vertices is inspired by the representation of the node number in the binary system, where each coordinate direction is represented by one bit. Concerning the neighbor orientation, we have $o(f, c) = 0$ independent of f and c at any face inside a base cell.

Cuboids. The tables n^F, n^C, $\tilde{f}(f, c)$ and $\tilde{c}(f, c)$ for cell type cuboid are presented in Fig. A.2. Again, the vertex numbering follows the binary system, where each coordinate direction is represented by one bit. As for quadrilaterals, at any face inside a base cell the neighbor orientation is $o(f, c) = 0$ independent of f and c . In Sect. 3.4 we already noted that not all permutations of face vertices are permitted in (2.3), and that only those corresponding to rigid body motions of the face are allowed. We denote this set of permutations by $\Pi' \subset \Pi_{\#NF}$, and have $|\Pi'| = 8$. The admitted node permutations Π' are given in Table A.3.

Prisms. In order to build coarse grids of tetrahedra and cuboids, prism cells can be used. A prism cell can be constructed as the tensor-product of a triangular and a quadrilateral cell. All algorithms given in this paper can also be applied for this cell type, see Fig. A.4 for the tables. Note that triangular faces have to be distinguished from quadrilateral faces by the face number and treated differently. The neighbor orientation $o(f, c)$ does not depend on c and is $o(f) = 4$ on triangular faces and $o(f) = 0$ on quadrilateral faces.

f	$n_0^F(f)$	$n_1^F(f)$
0	0	1
1	1	3
2	0	2
3	2	3

(a) n^F

c	$n_0^C(c)$	$n_1^C(c)$	$n_2^C(c)$	$n_3^C(c)$
0	0	0,1	0,2	0,1,2,3
1	0,1	1	0,1,2,3	1,3
2	0,2	0,1,2,3	2	2,3
3	0,1,2,3	1,3	2,3	3

(b) n^C

f \ c	0	1	2	3
0	-1	-1	3	3
1	2	-1	2	-1
2	-1	1	-1	1
3	0	0	-1	-1

(c) $\tilde{f}(f,c)$

f \ c	0	1	2	3
0	2	3	0	1
1	1	0	3	2
2	1	0	3	2
3	2	3	0	1

(d) $\tilde{c}(f,c)$

Fig. A.1: Tables for cell type quadrilateral.

f	$n_0^F(f)$	$n_1^F(f)$	$n_2^F(f)$	$n_3^F(f)$
0	0	1	2	3
1	0	1	4	5
2	0	2	4	6
3	1	3	5	7
4	2	3	6	7
5	4	5	6	7

(a) n^F

c	$n_0^C(c)$	$n_1^C(c)$	$n_2^C(c)$	$n_3^C(c)$	$n_4^C(c)$	$n_5^C(c)$	$n_6^C(c)$	$n_7^C(c)$
0	0	0,1	0,2	0,1,2,3	0,4	0,1,4,5	0,2,4,6	0,...,7
1	0,1	1	0,1,2,3	1,3	0,1,4,5	1,5	0,...,7	1,3,5,7
2	0,2	0,1,2,3	2	2,3	0,2,4,6	0,...,7	2,6	2,3,6,7
3	0,1,2,3	1,3	2,3	3	0,...,7	1,3,5,7	2,3,6,7	3,7
4	0,4	0,1,4,5	0,2,4,6	0,...,7	4	4,5	4,6	4,5,6,7
5	0,1,4,5	1,5	0,...,7	1,3,5,7	4,5	5	4,5,6,7	5,7
6	0,2,4,6	0,...,7	2,6	2,3,6,7	4,6	4,5,6,7	6	6,7
7	0,...,7	1,3,5,7	2,3,6,7	3,7	4,5,6,7	5,7	6,7	7

(b) n^C

f \ c	0	1	2	3	4	5	6	7
0	-1	-1	-1	-1	5	5	5	5
1	-1	-1	4	4	-1	-1	4	4
2	-1	3	-1	3	-1	3	-1	3
3	2	-1	2	-1	2	-1	2	-1
4	1	1	-1	-1	1	1	-1	-1
5	0	0	0	0	-1	-1	-1	-1

(c) $\tilde{f}(f,c)$

f \ c	0	1	2	3	4	5	6	7
0	4	5	6	7	0	1	2	3
1	2	3	0	1	6	7	4	5
2	1	0	3	2	5	4	7	6
3	1	0	3	2	5	4	7	6
4	2	3	0	1	6	7	4	5
5	4	5	6	7	0	1	2	3

(d) $\tilde{c}(f,c)$

Fig. A.2: Tables n^F, n^C, $\tilde{f}(f,c)$ and $\tilde{c}(f,c)$ for cell type cuboid.

	nc^F			
	E'_0	E'_1	E'_2	E'_3
$c'_0(\sigma, f)$	E_0	E_0, E_1	E_0, E_2	E_0, E_1, E_2, E_3
$c'_1(\sigma, f)$	E_0, E_1	E_1	E_0, E_1, E_2, E_3	E_1, E_3
$c'_2(\sigma, f)$	E_0, E_2	E_0, E_1, E_2, E_3	E_2	E_2, E_3
$c'_3(\sigma, f)$	E_0, E_1, E_2, E_3	E_1, E_3	E_2, E_3	E_3

(a) Table nc^F and child numbers c'_i.

	0	1	2	3
π_0	0	1	2	3
π_1	2	0	3	1
π_2	3	2	1	0
π_3	1	3	0	2

	0	1	2	3
π_4	1	0	3	2
π_5	3	1	2	0
π_6	2	3	0	1
π_7	0	2	1	3

(b) Orientation permutations π_k.

Fig. A.3: Tables for cuboid or prism (quadrilateral face).

f	$n_0^F(f)$	$n_1^F(f)$	$n_2^F(f)$	$n_3^F(f)$
0	1	2	4	5
1	2	0	5	3
2	0	1	3	4
3	0	1	2	-
4	3	4	5	-

(a) n^F

c	$n_0^C(c)$	$n_1^C(c)$	$n_2^C(c)$	$n_3^C(c)$	$n_4^C(c)$	$n_5^C(c)$
0	1,2	0,2	0,1	1,2,4,5	0,2,3,5	0,1,3,4
1	0	0,1	0,2	0,3	0,1,3,4	0,2,3,5
2	0,1	1	1,2	0,1,3,4	1,4	1,2,4,5
3	0,2	1,2	2	0,2,3,5	1,2,4,5	2,5
4	1,2,4,5	0,2,3,5	0,1,3,4	4,5	3,5	3,4
5	0,3	0,1,3,4	0,2,3,5	3	3,4	3,5
6	0,1,3,4	1,4	1,2,4,5	3,4	4	4,5
7	0,2,3,5	1,2,4,5	2,5	3,5	4,5	5

(b) n^C

f \ c	0	1	2	3	4	5	6	7
0	0	0	-1	-1	0	0	-1	-1
1	1	-1	1	-1	1	-1	1	-1
2	2	-1	-1	2	2	-1	-1	2
3	-1	-1	-1	-1	4	4	4	4
4	3	3	3	3	-1	-1	-1	-1

(c) $\tilde{f}(f, c)$

f \ c	0	1	2	3	4	5	6	7
0	1	0	3	2	5	4	7	6
1	2	3	0	1	6	7	4	5
2	3	2	1	0	7	6	5	4
3	4	5	6	7	0	1	2	3
4	4	5	6	7	0	1	2	3

(d) $\tilde{c}(f, c)$

Fig. A.4: Tables n^F, n^C, $\tilde{f}(f, c)$ and $\tilde{c}(f, c)$ for cell type prism.

REFERENCES

[1] D. Arnold, F. Brezzi, B. Cockburn, and D. Marini, *Unified analysis of discontinuous Galerkin methods for elliptic problems.* SIAM J. Numer. Anal., 39(5) (2002), pp. 1749–1779.

[2] F. Bassi and S. Rebay, *High-order accurate discontinuous finite element solution of the 2D Euler equations.* J. Comput. Phys. 138(2) (1997), pp. 251–285.

[3] P. Bastian, *Parallele Adaptive Mehrgitterverfahren.* Teubner Skripten zur Numerik, B. G. Teubner, Wiesbaden, 1996.

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

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

[6] K. Brix, M. Campos Pinto, and W. Dahmen, *A Multilevel Preconditioner for the Interior Penalty Discontinuous Galerkin Method.* SIAM J. Numer. Anal., 46 (2008), pp. 2742–2768.

[7] K. Brix, M. Campos Pinto, W. Dahmen, and R. Massjung, *Multilevel Preconditioners for the Interior Penalty Discontinuous Galerkin Method II - Quantitative Studies.* Commun. Comput. Phys., 5 (2009), pp. 296–325.

[8] K. Brix, S. Mogosan, S. Müller, and G. Schieffer, *Parallelisation of Multiscale-Based Grid Adaptation using Space-Filling Curves.* IGPM Report # 299, RWTH Aachen, 2009.

[9] P. G. Ciarlet, *The finite element method for elliptic problems.* North-Holland, Amsterdam, 1978.

[10] B. Cockburn and C.-W. Shu, *Runge-Kutta Discontinuous Galerkin methods for Convection-Dominated Problems.* J. Sci. Comput., 16(3) (2001), pp. 173–261.

[11] B. Cockburn, G.E. Karniadakis and C.-W. Shu, *Discontinuous Galerkin Methods.* Lecture Notes in Computational Science and Engineering, Vol. 11, Springer, Heidelberg, 2000.

[12] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 2001.

[13] W. Dahmen, *Wavelet and Multiscale Methods for Operator Equations.* Acta Numerica 6 (1997), pp. 55–228.

[14] G. Fekete, *Rendering and managing spherical data with sphere quadtrees.* Proceedings of the First IEEE Conference on Visualization (Visualization '90), IEEE computer Society Press, Los Alamitos (1990), pp. 176–186.

[15] A. Gill, *Introduction to the Theory of Finite-state Machines.* McGraw-Hill, New York, 1962.

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

[17] B. Grünbaum, *Convex Polytopes.* Springer, Heidelberg, 2nd edition, 2003.

[18] N. M. Josuttis, *The C++ Standard Library: A Tutorial and Reference.* Addison-Wesley, Boston, MA, 1999.

[19] F. Keinert, *Wavelets and Multiwavelets.* Chapman and Hall/CRC Press, Boca Raton, FL, 2004.

[20] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Boston, MA, 2nd edition, 1997.

[21] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks and Techniques; Binary Decision Diagrams.* Addison-Wesley, Boston, MA, 2009.

[22] Z. Kohavi, *Switching and Finite Automata Theory.* McGraw-Hill, New York, 1978.

[23] M. Lee, L. De Floriani, and H. Samet, *Constant-time neighbor finding in hierarchical tetrahedral meshes.* International Conference on Shape Modeling and Applications (SMI 2001), IEEE computer Society Press, Los Alamitos, CA (2001), pp. 286–295.

[24] M. Lee and H. Samet, *Navigating through Triangle Meshes Implemented as Linear Quadtrees.* ACM Trans. Graphics, 19 (2000), pp. 79–121.

[25] R. Massjung, *An hp-Error Estimate for an Unfitted Discontinuous Galerkin Method Applied to Elliptic Interface Problems.* IGPM Report # 300, RWTH Aachen, 2009.

[26] A. Schmidt and K. G. Siebert, *Design of Adaptive Finite Element Software.* Lecture Notes in Computational Science and Engineering, Vol. 42, Springer, Heidelberg, 2004.

[27] B. E. Smith and M. T. Johnson, *Programming the Intel 80386.* Scott Foresman, Glenview, IL, 1987.

[28] S. P. Vingron *Switching Theory: Insight Through Predicate Logic.* Springer, Heidelberg, 2004.

[29] A. Voss, *Notes on Adaptive Grids in 2D and 3D, Part I: Navigating through Cell Hierarchies using Cell Identifiers.* IGPM Report # 268, RWTH Aachen, 2006.

[30] G. Zumbusch, *Parallel multilevel methods. Adaptive mesh refinement and loadbalancing.* B. G. Teubner, Wiesbaden, 2003.