Notes on Adaptive Grids in 2D and 3D
Part I: Navigating through Cell Hierarchies using Cell Identifiers

Alexander Voß

Bericht Nr. 268 **Dezember 2006**

Key words: Data structure, adaptive grid, hybrid grid, cell enumeration strategy.
AMS subject classification: 68P05, 68P10, 65M50.

Institut für Geometrie und Praktische Mathematik
RWTH Aachen
Templergraben 55, D–52056 Aachen (Germany)

Notes on Adaptive Grids in 2D and 3D Part I: Navigating through Cell Hierarchies using Cell Identifiers

Alexander Voß

December 6, 2006

Abstract

In this paper we show how cells of different types from an adaptive grid in two or three spatial dimensions can be addressed by using so-called *cell identifiers* instead of memory consuming and error-prone pointer structures. Here we focus on horizontal and vertical grid navigation – the backbone of multilevel PDE-solvers based on discretizations that only need element connectivity across faces. That means, we present very fast and effective methods to *compute* cell identifiers of parent, children and neighbor cells based on a given identifier as well as appropriate *refinement strategies* for cells of type *triangles, rectangles, tetrahedrons, cubes and prisms*.

1 Introduction

A typical multidimensional grid, suitable for numerical computations, is composed of cells of one or multiple types. Usually, these cells just satisfy problem specific constraints such as limited side ratios, but using them in adaptive schemes requires further properties.

First of all, operations based on cells such as integration and function decomposition or refining and coarsening should be feasible – for both, the cell management and the numerical scheme. This works best for simple cell types like *triangles* or *rectangles* in two spatial dimensions or *tetrahedrons*, *cubes* and *prisms* in three dimensions. These cell types have in common that refining by bisection of edges yields geometrically similar refined cells of same cell type[1], which allows simple multilevel structures.

[1] For all cell types except the inner cells of a tetrahedron

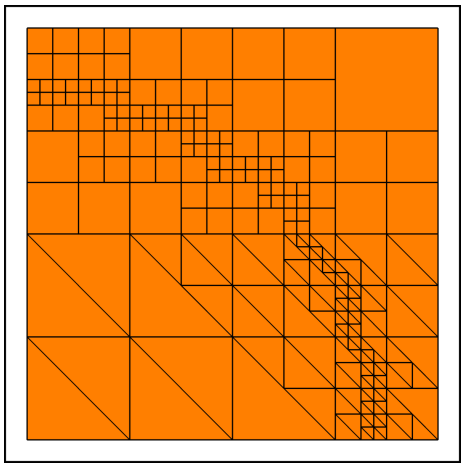

Figure 1: An adaptive grid composed of triangles and rectangles.

Moreover, the grid should initially be constructed without specific knowledge of the numerical solution to be computed. The adaption strategy itself is responsible for refining or coarsening cells at run-time in order to obtain a given accuracy with minimal effort.

From both we conclude the following setting to be valid throughout this paper:

- The initial grid is composed of few cells, called *base cells* (number #`BC`).
- These base cells are triangles, rectangles, tetrahedrons, cubes or prisms. It is not necessary, that all grid cells are of the same type.
- Each cell is either a base cell or a cell resulting from a refinement process of a base cell, hence a part of a base cell.

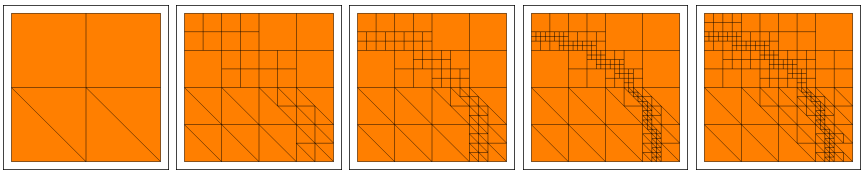

Figure 2: Example showing base cells (left) and various refinement steps.

Typical numerical schemes for differential equations such as finite volume schemes calculate cell data, explicitly or implicitly, by using information taken from neighboring cells – this involves *horizontal grid navigation*, which means the possibility to access the neighboring cells of a given cell on the same level. Adaptive strategies, however, always involve parent–child relations of cells, which is a *vertical grid navigation*.

One way to implement neighboring as well as parent–child relationships is to connect cells with so-called *pointers*, which is, in most cases, the memory address of a cell. A brief calculation of the number of pointers required to hold

these relationships for each cell in three dimensions yields one pointer for the parent cell, a pointer for each child cell and a neighbor pointer for each cell face. Using this approach a tetrahedron with only four faces requires 13 pointers, just for the cell management.

We suggest another idea to realize the cell navigation required. We equip each possible cell with a *unique cell identifier* (`id`) and generally access the corresponding cell data by means of an *associative mapping*, i.e., this map determines a pointer to the cell data from a given `id`. We then perform the horizontal and vertical navigation by *calculating* parent, neighboring and children cell identifiers from a given identifier, in contrast to use memory. This is possible as most cells stem from a refinement process, hence being a child cell of some base cell. The grid navigation algorithms can be seen as a virtual connectivity amongst potential cells of the hierarchy.

One price we have to pay is the amount of memory we need for the associative mapping. But this is much less than the one needed for the number of pointers from the first approach. Additionally, the mapping is realized by means of hash functions and thus very fast, as are also the identifier algorithms we provide in this paper.

Before we introduce the cell identifier `id` and the navigation algorithms, we would like to present the supported cell types along with appropriate refinement strategies. This is done in Section 2. Then we build up a cell identifier in Section 3 and show, how to efficiently calculate cell identifiers for parents, children and neighbors in Section 4. As we will see, parent and child identifiers are pretty easy, the most crucial point is the *neighboring algorithm*, being the heart of this paper.

2 Cell Types

Supported cell types are

- *triangles* (Table 1, Figure 3) and
- *rectangles* (Table 2, Figure 5),

in two spatial dimensions, and

- *tetrahedrons* (Table 3, Figure 7),
- *cubes* (Table 4, Figure 9) and
- *prisms* (Table 5, Figure 11)

in three dimensions.

For each cell type we have to promote one way

- to *enumerate* cell nodes, cell faces, and children cells, and
- to *refine* such a cell.

Subsequently, we need the following definitions:

$\#\mathtt{N}$	number of nodes per cell type;
$\mathtt{N}$	a cell node with $\mathtt{N} \in \{\mathtt{N}_0, \ldots, \mathtt{N}_{\#\mathtt{N}-1}\}$ and $\mathtt{N}_i \in \mathbb{R}^2$ or $\mathtt{N}_i \in \mathbb{R}^3$;
$\mathtt{N}_{i_1 \ldots i_n}$	an intermediate node with $\mathtt{N}_{i_1 \ldots i_n} = \sum_j \mathtt{N}_{i_j}/n$ (cf. Figure 3);
$\#\mathtt{F}$	number of faces per cell type;
$\mathtt{F}$	a face with $\mathtt{F} \in \{0, \ldots, \#\mathtt{F}-1\}$;
$\#\mathtt{C}$	number of children per cell type;
$\mathtt{C}$	a child or child-`id` with $\mathtt{C} \in \{0, \ldots, \#\mathtt{C}-1\}$;
$\mathtt{T}$	a substitution table with $\mathtt{T} \in \{0, \ldots, \#\mathtt{F}(\mathtt{C})-1\}$;
$\mathtt{N}(\mathtt{F})$	index vector of nodes per face, e.g., $\mathtt{N}(\mathtt{F}=0) = [1,2]$ means that face $\mathtt{F}=0$ is defined by the nodes $\mathtt{N}_1$ and $\mathtt{N}_2$;
$\mathtt{N}(\mathtt{C})$	index vector of nodes per child, e.g., $\mathtt{N}(\mathtt{C}=1) = [0, 01, 02]$ means that child $\mathtt{C}=1$ is defined by the nodes $\mathtt{N}_0$, $\mathtt{N}_{01}$ and $\mathtt{N}_{02}$;
$\mathtt{F}(\mathtt{C})$	inner faces per child, e.g., $\mathtt{F}(\mathtt{C}=2) = [1]$ means that the only inner face of child $\mathtt{C}=2$ is its face 1;
$\mathtt{T}(\mathtt{C})$	index vector of substitution tables per child, e.g. $\mathtt{T}(\mathtt{C}=2) = [1]$ means that the only substitution table for child $\mathtt{C}=2$ is table 1;
$\mathtt{S}(\mathtt{T})$	substitution table, e.g., $\mathtt{S}(\mathtt{T}=0) = [1,0,3,2]$ means that we replace as follows: $0 \to 1$, $1 \to 0$, $2 \to 3$, $3 \to 2$.

If we address neighbor i we refer to the cell adjacent to the i'th face. The index vector of nodes per child $\mathtt{N}(\mathtt{C})$ not only states the involved nodes to define the edges but also their order and orientation. Note also, that we have as many substitution tables as inner faces per child.

The following tables provide the specific characteristic values defined above per cell type. The use of tables $\mathtt{F}(\mathtt{C})$, $\mathtt{T}(\mathtt{C})$ and $\mathtt{S}(\mathtt{T})$ become clear in Section 4 in connection with the horizontal and vertical grid navigation.

2.1 Triangles

geometry	
$\#\mathtt{N}$	3
$\#\mathtt{F}$	3
$\#\mathtt{C}$	4

$\mathtt{F}$	$\mathtt{N}(\mathtt{F})$
0	$[1,2]$
1	$[0,2]$
2	$[0,1]$

$\mathtt{C}$	$\mathtt{N}(\mathtt{C})$	$\#\mathtt{F}(\mathtt{C})$	$\mathtt{F}(\mathtt{C})$	$\mathtt{T}(\mathtt{C})$
0	$[12, 02, 01]$	3	$[0,1,2]$	$[0,1,2]$
1	$[0, 01, 02]$	1	$[0]$	$[0]$
2	$[01, 1, 12]$	1	$[1]$	$[1]$
3	$[02, 12, 2]$	1	$[2]$	$[2]$

$\mathtt{T}$	$\mathtt{S}(\mathtt{T})$
0	$[1,0,3,2]$
1	$[2,3,0,1]$
2	$[3,2,1,0]$

Table 1: Triangle characteristics.

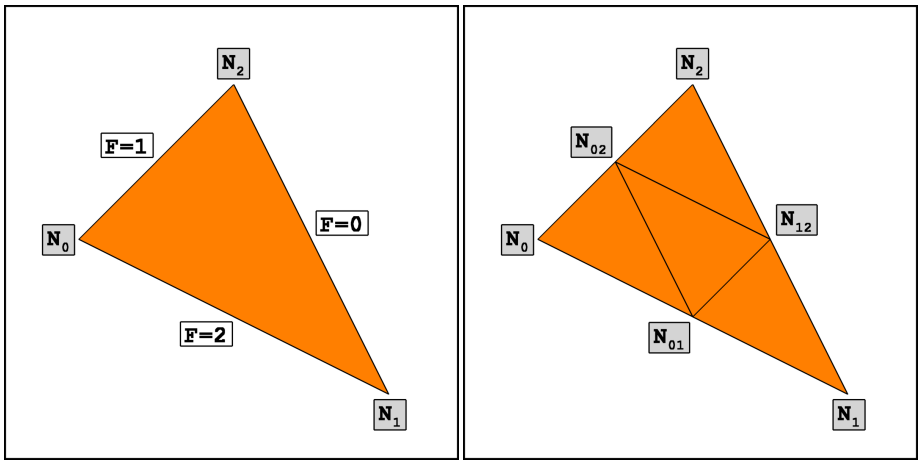

Figure 3: Triangle nodes and faces.

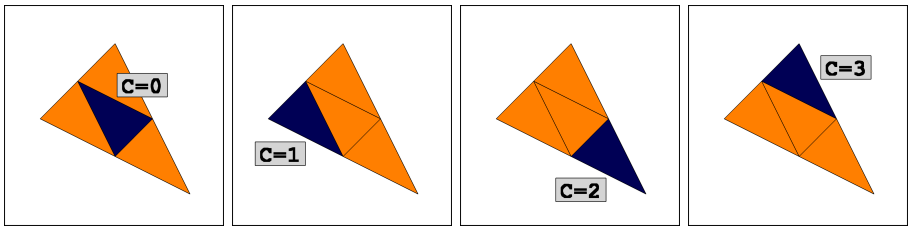

Figure 4: Triangle refinement.

2.2 Rectangles

geometry	
#N	4
#F	4
#C	4

F	N(F)
0	[0, 1]
1	[1, 3]
2	[0, 2]
3	[2, 3]

C	N(C)	#F(C)	F(C)	T(C)
0	[0, 01, 02, 0123]	2	[1, 3]	[0, 1]
1	[01, 1, 0123, 13]	2	[2, 3]	[0, 1]
2	[02, 0123, 2, 23]	2	[0, 1]	[1, 0]
3	[0123, 13, 23, 3]	2	[0, 2]	[1, 0]

T	S(T)
0	[1, 0, 3, 2]
1	[2, 3, 0, 1]

Table 2: Rectangle characteristics.

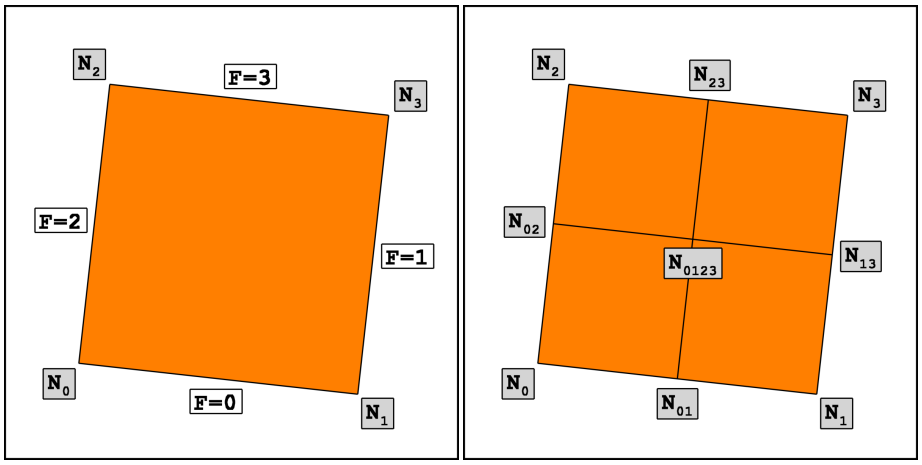

Figure 5: Rectangle nodes and faces.

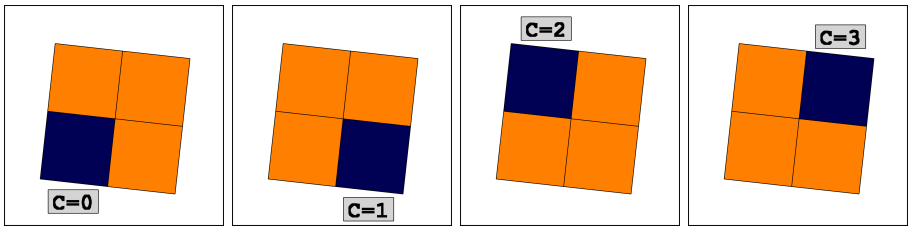

Figure 6: Rectangle refinement.

2.3 Tetrahedrons

geometry	
#N	4
#F	4
#C	8

F	N(F)
0	[0, 1, 2]
1	[0, 1, 3]
2	[0, 2, 3]
3	[1, 2, 3]

C	N(C)	#F(C)	F(C)	T(C)
0	[12, 02, 01, 03]	3	[1, 2, 3]	[0, 1, 2]
1	[13, 03, 12, 01]	3	[0, 3, 2]	[3, 4, 5]
2	[23, 12, 03, 02]	3	[0, 3, 1]	[6, 7, 8]
3	[03, 23, 13, 12]	3	[1, 2, 0]	[9, 10, 11]
4	[0, 01, 02, 03]	1	[3]	[12]
5	[01, 1, 12, 13]	1	[2]	[13]
6	[02, 12, 2, 23]	1	[1]	[14]
7	[03, 13, 23, 3]	1	[0]	[15]

T	S(T)	T	S(T)
0	[2, 3, −, −, 5, 7, −, 6]	8	[−, 1, 6, −, 7, 5, −, 4]
1	[1, −, 3, −, 6, −, 7, 5]	9	[−, 0, −, 2, 6, 4, −, 5]
2	[4, −, −, 3, −, 6, 5, 7]	10	[−, −, 0, 1, 5, −, 4, 6]
3	[2, 3, −, −, 6, 4, 7, −]	11	[0, −, −, 7, 4, 6, 5, −]
4	[−, 0, −, 2, −, 7, 4, 6]	12	[−, −, −, 3, 0, 6, 5, 7]
5	[−, 5, 2, −, 7, −, 6, 4]	13	[−, −, 2, −, 7, 1, 6, 4]
6	[1, −, 3, −, 5, 7, 4, −]	14	[−, 1, −, −, 7, 5, 2, 4]
7	[−, −, 0, 1, −, 4, 7, 5]	15	[0, −, −, −, 4, 6, 5, 3]

Table 3: Tetrahedrons characteristics.

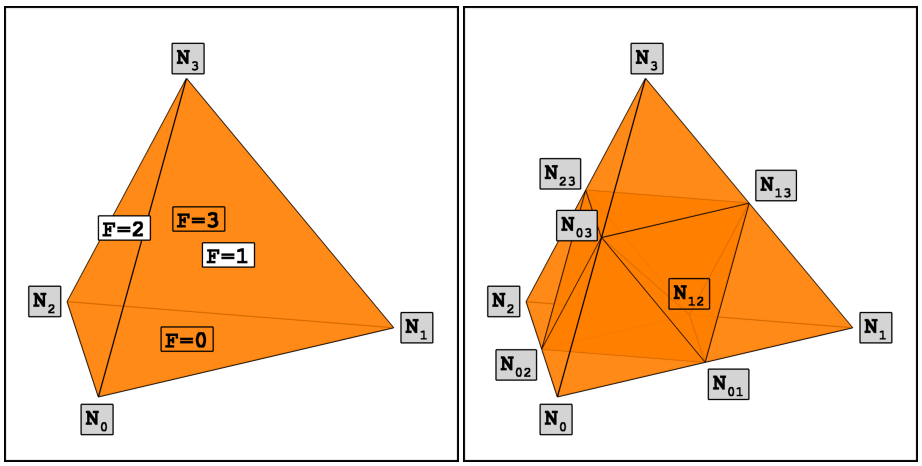

Figure 7: Tetrahedrons nodes and faces.

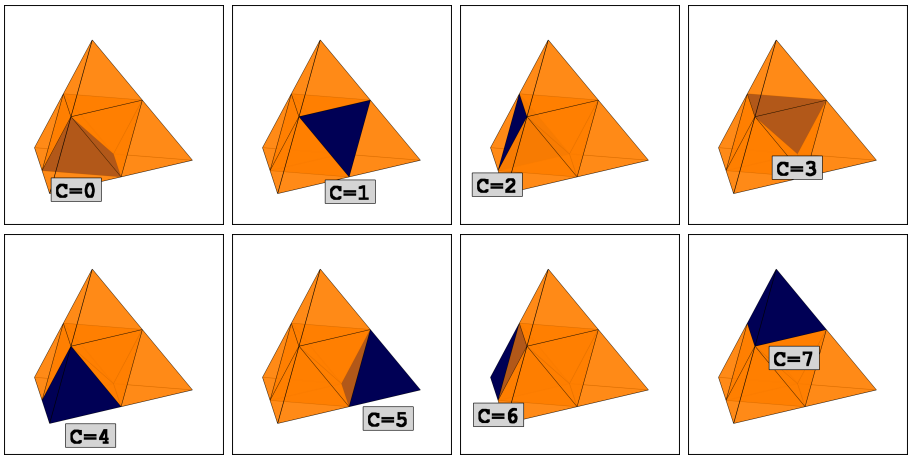

Figure 8: Tetrahedrons refinement.

2.4 Cubes

geometry	
#N	8
#F	6
#C	8

F	N(F)	F	N(F)
0	[0, 1, 2, 3]	3	[1, 3, 5, 7]
1	[0, 1, 4, 5]	4	[2, 3, 6, 7]
2	[0, 2, 4, 6]	5	[4, 5, 6, 7]

T	S(T)
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]

C	N(C)	#F(C)	F(C)	T(C)
0	[0, 01, 02, 0123, 04, 0145, 0246, 0-7]	3	[3, 4, 5]	[2, 1, 0]
1	[01, 1, 0123, 13, 0145, 15, 0-7, 1357]	3	[2, 4, 5]	[2, 1, 0]
2	[02, 0123, 2, 23, 0246, 0-7, 26, 2367]	3	[1, 3, 5]	[1, 2, 0]
3	[0123, 13, 23, 3, 0-7, 1357, 2367, 37]	3	[1, 2, 5]	[1, 2, 0]
4	[04, 0145, 0246, 0-7, 4, 45, 46, 4567]	3	[0, 3, 4]	[0, 2, 1]
5	[0145, 15, 0-7, 1357, 45, 5, 4567, 57]	3	[0, 2, 4]	[0, 2, 1]
6	[0246, 0-7, 26, 2367, 46, 4567, 6, 67]	3	[0, 1, 3]	[0, 1, 2]
7	[0-7, 1357, 2367, 37, 4567, 57, 67, 7]	3	[0, 1, 2]	[0, 1, 2]

Table 4: Cubes characteristics.

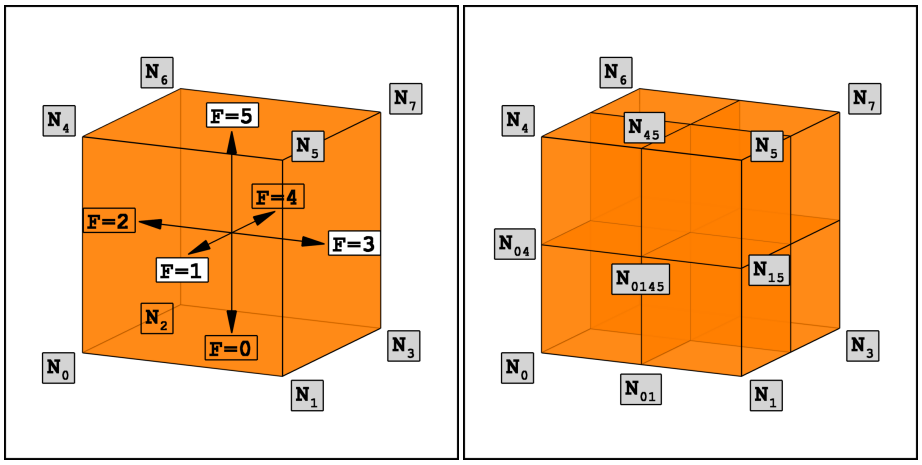

Figure 9: Cubes nodes and faces.

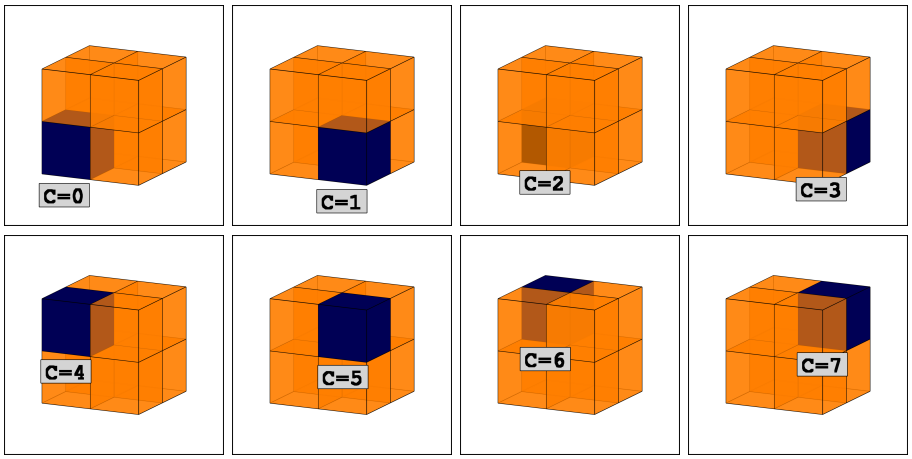

Figure 10: Cubes refinement.

2.5 Prisms

geometry	
#N	6
#F	5
#C	8

F	N(F)
0	[0, 1, 2]
1	[3, 4, 5]
2	[0, 1, 3, 4]
3	[0, 2, 3, 5]
4	[1, 2, 4, 5]

T	S(T)
0	[3, 4, 5, 0, 1, 2, 7, 6]
1	[6, 2, 1, 7, 5, 4, 0, 3]
0	[2, 6, 0, 5, 7, 3, 1, 4]
1	[1, 0, 6, 4, 3, 7, 2, 5]

C	N(C)	#F(C)	F(C)	T(C)
0	[0, 01, 02, 03, 0134, 0235]	2	[1, 4]	[0, 1]
1	[01, 1, 12, 0134, 14, 1245]	2	[1, 3]	[0, 2]
2	[02, 12, 2, 0235, 1245, 25]	2	[1, 2]	[0, 3]
3	[03, 0134, 0235, 3, 34, 35]	2	[0, 4]	[0, 1]
4	[0134, 14, 1245, 34, 4, 45]	2	[0, 3]	[0, 2]
5	[0235, 1245, 25, 35, 45, 5]	2	[0, 2]	[0, 3]
6	[12, 02, 01, 1245, 0235, 0134]	4	[1, 2, 3, 4]	[0, 3, 2, 1]
7	[1245, 0235, 0134, 45, 35, 34]	4	[0, 2, 3, 4]	[0, 3, 2, 1]

Table 5: Prisms characteristics.

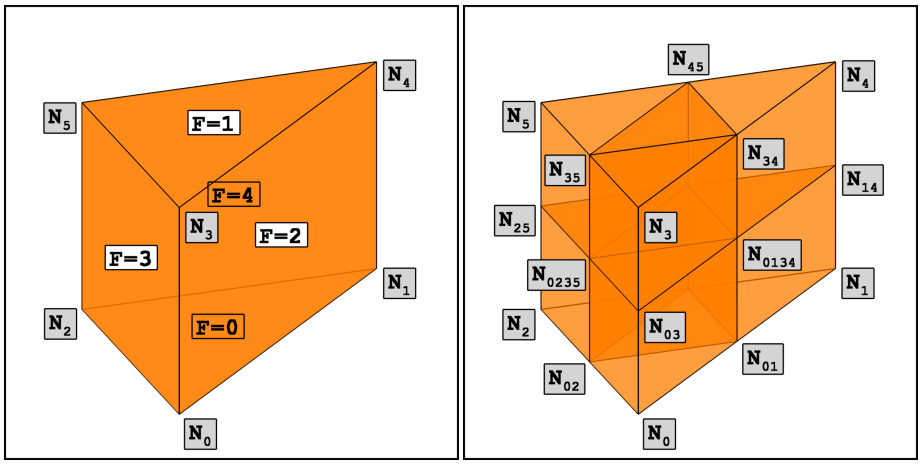

Figure 11: Prisms nodes and faces.

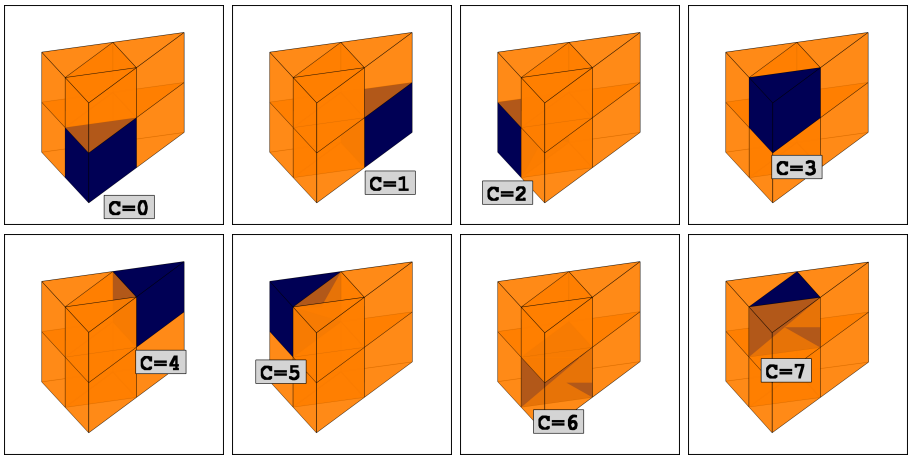

Figure 12: Prisms refinement.

3 Cell Identifier

As said before, we assume the initial grid to be composed of (base) cells of compatible types; see Figure 1 for instance. The grid changes during run-time due to the refinement and coarsening operations only. Hence, all cells other than base cells are children of a specific base cell or children of another child cell.
We now assemble all information necessary to identify each possible grid cell, i.e. for all possible refinement levels up to a maximum level `L`, to a single unique identifier `id`.

The key idea is to collect the sequence of the refinement steps, i.e., the sequence of child numbers `C`, up to the point when we reach the destination cell. Consider the following sequence of child numbers:

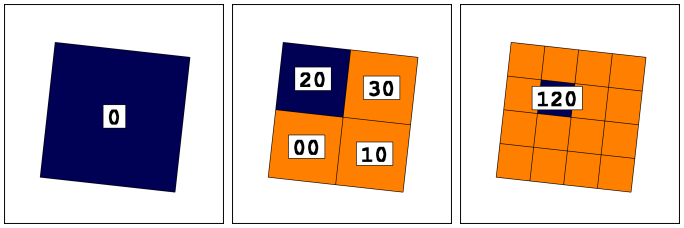

Figure 13: Refinement steps.

We start with the base cell 0 on level 0, refine this base cell and obtain the four children cells 00,10,20 and 30 on level 1. Then we focus on cell 20, refine and identify cell 120 on level 2.
Such a sequence of chosen children on different levels up to a level `L` is called *path*, written as a list of digits $\mathtt{P} = \mathtt{C}_{\mathtt{L}} \ldots \mathtt{C}_0$. In the example above we stop the refinement at the maximum level $\mathtt{L} = \mathtt{L}_{\max} = 2$.
Together with the cell *type* `T`, the refinement *level* `L` and the underlying *base*

cell BC the id looks

$$\texttt{id} = * \,|\, \texttt{T} \,|\, \texttt{BC} \,|\, \texttt{L} \,|\, \texttt{P},$$

with a separator | (for readability only). The $*$ marks a reserved region for additional information, e.g., cluster information for multiprocessor applications or some flags; we exclude this from the discussion here.
As a convention we define the base cell to live on level 0.

3.1 Bit Gymnastics

Consider a two-dimensional initial grid composed of not more than 65536 base cells of triangular or rectangular type. If we assume a maximum level $\texttt{L}_{\max}$ not greater than 15, then the resulting id needs

information	bits
cell type T	1,
base cell BC	16,
level L	4,
path P (dimension times $\texttt{L}_{\max}$)	32,
reserved $*$	11,
sum	64.

For a 32-bit computer this means the id needs as much space as two pointers, for a modern 64-bit one it is only one pointer.
If we look at a three-dimensional grid composed of tetrahedrons, cubes and prisms, we obtain

information	bits
cell type T	2,
base cell BC	16,
level L	4,
path P (dimension times $\texttt{L}_{\max}$)	48,
reserved $*$	26,
sum	96.

Obviously, it is easy to increase the number of supported levels or base cells by distributing the reserved bits appropriately[2].

4 Navigation

Given an id we now show how to calculate the ids of the parent[3], the children and the neighbors. Here the algorithm of the neighbors is the crucial part. At the end of this section we complete this approach by calculating neighbors living in different base cells.

In the following we start with a given id of a specific base cell BC of a specific type T. As the base cell and the type is fixed for now, we skip these information and any leading zeros of the path and display the ids as a tuple of level and path: $\texttt{id} = \texttt{L} \,|\, \texttt{P}$ or $\texttt{id} = \texttt{L} \,|\, \texttt{C}_\texttt{L} \ldots \texttt{C}_0$. In some cases an operation does not yield a valid result. Then we indicate this by using INV for an invalid id.

[2]Note, that modern computers *align* memory. That means, they allocate memory of a multiple of 32 bit size for data structures anyway.
[3]Yes, only one ...

4.1 Parent

The parent of $\texttt{id} = \texttt{L} \,|\, \texttt{C}_\texttt{L} \ldots \texttt{C}_0$ is given by

$$\texttt{parent}(\texttt{id}) = \begin{cases} \texttt{INV} & \texttt{L} = 0 \\ \texttt{L}-1 \,|\, 0\,\texttt{C}_{\texttt{L}-1} \ldots \texttt{C}_0 & \texttt{L} > 0. \end{cases}$$

4.2 Children

The i'th child of $\texttt{id} = \texttt{L} \,|\, \texttt{C}_\texttt{L} \ldots \texttt{C}_0$ is given by

$$\texttt{child}_i(\texttt{id}) = \begin{cases} \texttt{INV} & \texttt{L} = \texttt{L}_{\max} \\ \texttt{L}+1 \,|\, i\,\texttt{C}_\texttt{L} \ldots \texttt{C}_0 & \texttt{L} < \texttt{L}_{\max}. \end{cases}$$

4.3 Neighbors

The algorithm for computing the neighbor ids of an id for all neighbor cells lying in the same base cell is best motivated by an example. To this end we consider a rectangular base cell with cells on level 1, see left picture in Figure 14. We want to sort the neighbor-ids the same way as we sort the faces, namely such that the i'th neighbor is on the i'th face.

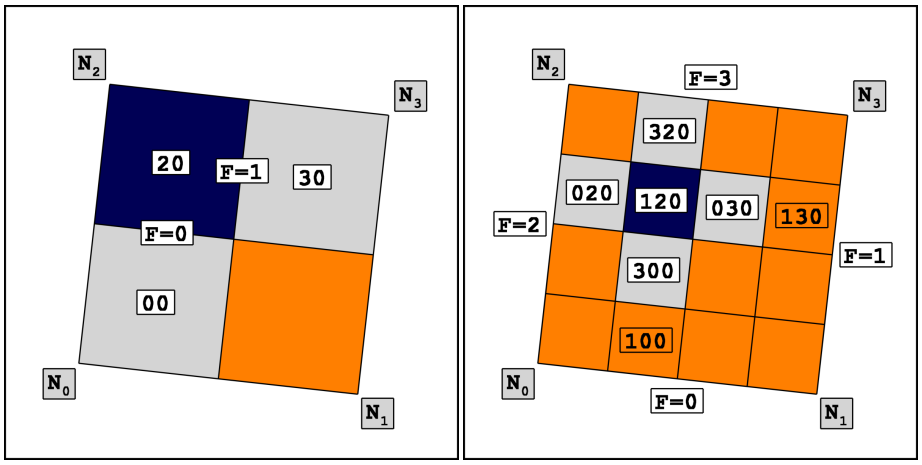

Figure 14: Neighbors on different levels.

The cell with $\texttt{id} = 1|20$ has only two neighbors on the same level: $1|00$ on face 0 and $1|30$ on face 1; that's it. In general, all rectangular cells have exactly two neighbors on the same level with same parent cell, all further neighbors have a different parent cell, cf. also Table 2.
Now we increase the level and consider the cell $\texttt{id} = 2|120$ on level 2, see again Figure 14, right picture. The two neighbors on level 2 with same parent are: $2|020$ at face 2 and $2|320$ at face 3. Obviously, the missing neighbors live in neighboring cells of the parent cell $1|20$, i.e., in $1|00$ at face 0 and in $1|30$ at face 1.
We now consider "the same" child on such a neighbor cell, for instance $2|130$, which is obviously not the neighbor we search. But if we switch side in the opposite direction we took the neighbor of the parent, it is.

Doing the same for the parent neighbor at face 0, we first consider the cell 2|100. Switching side in the opposite direction leads to 2|300, which is the last missing neighbor of 2|120 at face 0.

What have we done? After calculating the immediate neighbors within the parent cell, we considered neighbors of the parent cell. Starting from the cell representing the same child in the neighbor cells we switched side in opposite direction, or, more general, substitute the child by another child being the neighbor child at the specific face.
In the example above we replaced digit 1 by 0 for cell 2|130 in order to obtain the neighbor 2|030, and 1 by 3 for cell 2|100 in order to obtain 2|300, respectively. Here we found all neighbors after taking the parent neighbors into account. In general, all this has to be done until we get a neighbor at each face.

Technically speaking, we exchanged $\mathtt{C}_j$ on different levels in the path of the given `id`, depending on the face. This is what the substitution tables are for. They depend on the child and face under consideration. This replacement strategy is formally the same for all cell-types, where specific replacement-rules are explained below.

We now formulate the steps above for the example $\mathtt{id} = 2|120$ in terms of the final algorithm. Characteristic values used can be found in Table 2. The general algorithm, valid for all supported cell types, is given subsequently.

- Start: $\mathtt{id} = \mathtt{L}|\mathtt{C}_\mathtt{L} \dots \mathtt{C}_0 = 2|120$.
- Traverse all children on all levels downwards in level (left to right):
  for $\mathtt{C}$ in $[120]$:
  1. Child on current level: $\mathtt{C} = 1$, level 2.
  2. Determine faces (possible neighbors) and table indices for current child: $\mathtt{F}(\mathtt{C}) = [2, 3]$, $\mathtt{T}(\mathtt{C}) = [0, 1]$.
  3. Traverse all faces for which the neighbor `id` is not known yet:
     for $\mathtt{F}$ in $[2, 3]$ with $\mathtt{T}$ in $[0, 1]$:
     (a) Determine substitution table for $\mathtt{F} = 2$, $\mathtt{T} = 0$: $\mathtt{S} = [1, 0, 3, 2]$.
     (b) Replace $\mathtt{C}_j$ on all levels upwards according to the substitution table $\mathtt{S}$:
     for $j \in \{l = 2, \dots, \mathtt{L} = 2\}$ replace $\mathtt{C}_j$ with $\mathtt{S}(\mathtt{C}_j)$:
     $j = 2$: $\mathtt{C}_2 = \mathtt{S}(\mathtt{C}_2 = 1) = 0$
     (c) Neighbor on $\mathtt{F} = 2$ found: $2|020$.
     (a) Determine substitution table for $\mathtt{F} = 3$, $\mathtt{T} = 1$: $\mathtt{S} = [2, 3, 0, 1]$.
     (b) Replace $\mathtt{C}_j$ on all levels upwards according to the substitution table $\mathtt{S}$:
     for $j \in \{l = 2, \dots, \mathtt{L} = 2\}$ replace $\mathtt{C}_j$ with $\mathtt{S}(\mathtt{C}_j)$:
     $j = 2$: $\mathtt{C}_2 = \mathtt{S}(\mathtt{C}_2 = 1) = 3$
     (c) Neighbor on $\mathtt{F} = 3$ found: $2|320$.
  1. Child on current level: $\mathtt{C} = 2$, level 1.
  2. Determine faces (possible neighbors) and table indices for current child: $\mathtt{F}(\mathtt{C}) = [0, 1]$, $\mathtt{T}(\mathtt{C}) = [1, 0]$.

3. Traverse all faces for that the neighbor `id` at that face is not already known:
   for `F` in $[0, 1]$ with `T` in $[1, 0]$:
   (a) Determine substitution table for $\texttt{F} = 0$, $\texttt{T} = 1$: $\texttt{S} = [2, 3, 0, 1]$.
   (b) Replace $\texttt{C}_j$ on all levels upwards according to the substitution table `S`:
   for $j \in \{l = 1, \ldots, \texttt{L} = 2\}$ replace $\texttt{C}_j$ with $\texttt{S}(\texttt{C}_j)$:
   $j = 1$: $\texttt{C}_1 = \texttt{S}(\texttt{C}_1 = 2) = 0$
   $j = 2$: $\texttt{C}_2 = \texttt{S}(\texttt{C}_2 = 1) = 3$
   (c) Neighbor on $\texttt{F} = 0$ found: $2|300$.
   (a) Determine substitution table for $\texttt{F} = 1$, $\texttt{T} = 0$: $\texttt{S} = [1, 0, 3, 2]$.
   (b) Replace $\texttt{C}_j$ on all levels upwards according to the substitution table `S`:
   for $j \in \{l = 1, \ldots, \texttt{L} = 2\}$ replace $\texttt{C}_j$ with $\texttt{S}(\texttt{C}_j)$:
   $j = 1$: $\texttt{C}_1 = \texttt{S}(\texttt{C}_1 = 2) = 3$
   $j = 2$: $\texttt{C}_2 = \texttt{S}(\texttt{C}_2 = 1) = 0$
   (c) Neighbor on $\texttt{F} = 1$ found: $2|030$.

- All neighbors were found.

The algorithm valid for *all* supported cell types reads:

- Traverse all children on all levels:
  for `C` in $\texttt{C}_\texttt{L} \ldots \texttt{C}_0$
  1. Child on current level l: $\texttt{C}_l$.
  2. Determine faces (possible neighbors) and table indices for current child: $\texttt{F}(\texttt{C})$, $\texttt{T}(\texttt{C})$.
  3. Traverse all faces for that the neighbor `id` at that face is not already known:
     for `F` in $\texttt{F}(\texttt{C})$ with `T` in $\texttt{T}(\texttt{C})$:
     (a) Determine substitution table for `F`, `T`: `S`.
     (b) Replace $\texttt{C}_j$ on all levels upwards according to the substitution table `S`:
     for $j \in \{l, \ldots, \texttt{L}\}$ replace $\texttt{C}_j$ with $\texttt{S}(\texttt{C}_j)$.
     (c) Neighbor at face `F` found.
     (d) Stop algorithm if all neighbors were found.
  4. All neighbors on level l were found.

- All neighbors were found or some neighbors live on adjacent base cells.

We like to emphasize, that even the algorithm may look complicated it is very fast as it is purely build on bit operations and look-up tables. Moreover, there exist optimized specific implementations for all cell types, realizing, for instance, simple substitution tables directly via algebraic operations or short loops via unrolling.

4.3.1 Neighbors in adjacent base cells

It may happen that the neighbor algorithm does not deliver neighbor `id`s at all faces. This happens, if and only if a cell is adjacent to a base cell's face. Then the existence of a neighbor cell depends on the fact whether the base cell has a neighbor in that direction or not. Figure 15 shows an example.

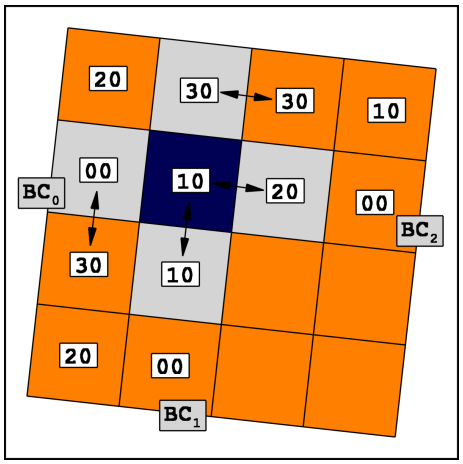

Figure 15: Example showing base cell neighbors.

In this example we composed four base cells with different orientations and show all relevant children cells on level 1. Note, the position of label $\mathtt{BC}_i$ simultaneously denotes $\mathtt{N}_0$ of the respective base cell.

We consider cell $\mathtt{BC}_0|1|10$ and its neighbors. First of all, two neighbor cells live inside $\mathtt{BC}_0$. They are determined by the neighbor algorithm. But then the algorithm stops and neighbors at face $\mathtt{F} = 0$ and $\mathtt{F} = 1$ are still missing – they live in $\mathtt{BC}_1$ and $\mathtt{BC}_2$.
The only information we need in order to know *all* neighbors at the base cell's face on *all* levels is a number of fixed substitution tables (per base cell pair) if we choose the refinement such that at each face always live the same children on every level. In the example above the substitutions read:

base cells	faces	substitutions
$\mathtt{BC}_0 \leftrightarrow \mathtt{BC}_1$	$0 \leftrightarrow 1$	$[3,1,2,0]$
$\mathtt{BC}_0 \leftrightarrow \mathtt{BC}_2$	$1 \leftrightarrow 3$	$[0,2,1,3]$

Thus, if the neighbor algorithm fails to determine some neighbors we take these additional substitution tables into account and replace each child $\mathtt{C}_i$ in the path according to the respective table. This immediately yields the missing neighbor cells.
We like to stress, that this is possible only if the refinement strategy is chosen appropriately. But then it works even for grids composed of different cell types. In our case, all refinement strategies work, i.e., one can combine all mentioned cell types of the same spatial dimension.

5 Outlook

This paper is the first part of a more global description of multidimensional grids based on cell identifiers. The sequels basically contain the grid management and grid libraries as well as applications in two and three dimensions.

6 Acknowledgment

We like to thank Prof. Dahmen, Dr. Massjung, Dr. Müller, Dr. Gottschlich-Müller and Mr. Brix for the fruitful discussions concerning this topic.

List of Tables

1 Triangle characteristics. 4
2 Rectangle characteristics. 5
3 Tetrahedrons characteristics. 7
4 Cubes characteristics. 8
5 Prisms characteristics. 10

List of Figures

1 An adaptive grid composed of triangles and rectangles. 2
2 Example showing base cells (left) and various refinement steps. . 2
3 Triangle nodes and faces. 5
4 Triangle refinement. 5
5 Rectangle nodes and faces. 6
6 Rectangle refinement. 6
7 Tetrahedrons nodes and faces. 7
8 Tetrahedrons refinement. 8
9 Cubes nodes and faces. 9
10 Cubes refinement. 9
11 Prisms nodes and faces. 10
12 Prisms refinement. 11
13 Refinement steps. 11
14 Neighbors on different levels. 13
15 Example showing base cell neighbors. 16