BASIS3,

A Data Structure for 3-Dimensional Sparse Grids

P.W. Hemker and P.M. de Zeeuw
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Abstract

In this report a data structure and basic procedures are described, that can be used for the implementation of adaptive sparse grid algorithms in three dimensions. The basic elements are rectangular cells (cubes and parallelepipeds) that fit in a tree-type data structure. A cell can be split –in three ways– in two equal smaller cells. In this way any grid cell in the structure can be refined. The data structure is completely symmetric with respect to the 3 space dimensions.

Basis routines to handle the data structure are provided. The same software can be used for the solution of 1- or 2-dimensional problems. In the appendix a PASCAL prototype implementation is given and also the available FORTRAN implementation is described.

1 The geometric structure

1.1 Coordinates, grids and cell elements

For the construction of discretisation schemes for the solution of PDEs, we may use physical and computational coordinates. In this report, for the description of the data structure, we only use computational coordinates in 3 space dimensions. We assume a Cartesian coordinate system, and for convenience we distinguish the x-, y- and z-coordinate directions. Further we identify in this coordinate system an origin and a unit length.

We will describe a data structure for handling adaptive sparse grids in 3 dimensions, both for finite element and for finite volume type discretisation methods. For the sparse grids we will need the simultaneous use of many different grids, cells, nodal points etc.. However, there is only one *basic grid*, $\mathcal{R}_{0,0,0}$. This is the regular rectangular grid in the 3-dimensional space, consisting of all nodal points in $\mathbb{R}^3$ that are located at points with all integer coordinates in the computational space. Hence

$$\mathcal{R}_{0,0,0} = \{(i,j,k)\in\mathbb{R}^3;\ i\in\mathbb{Z}, j\in\mathbb{Z}, k\in\mathbb{Z}\}\ . \tag{1}$$

Similarly, we introduce many (infinite) grids with nodal points at dyadic points in $\mathbb{R}^3$. For any $(l,m,n)\in\mathbb{Z}^3$ we introduce a grid $\mathcal{R}_{l,m,n} \subset \mathbb{R}^3$ as

$$\mathcal{R}_{l,m,n} = \{(i2^{-l}, j2^{-m}, k2^{-n})\in\mathbb{R}^3;\ i\in\mathbb{Z}, j\in\mathbb{Z}, k\in\mathbb{Z}\}\ . \tag{2}$$

We call (l, m, n) the *level-vector* of the grid. We also say that $\mathcal{R}_{l,m,n}$ is a grid *on the* $(l+m+n)$*-level.* The $\mathcal{R}_{l,m,n}$ in (2) are all possible grids. For an impression of the relation between these grids for $l + m + n \geq 0$, we refer to Figure 1. On these grids we may wish to handle all kinds of vertex- or box- centered discretisation methods, such as finite element, mixed finite element or finite volume methods. I.e. we may wish to associate numerical values with any kind of nodal point (=cell vertex), cell (or cell center), cell face or cell edge. Of course, in practice only finite parts, and a selection of all possibilities will be used.

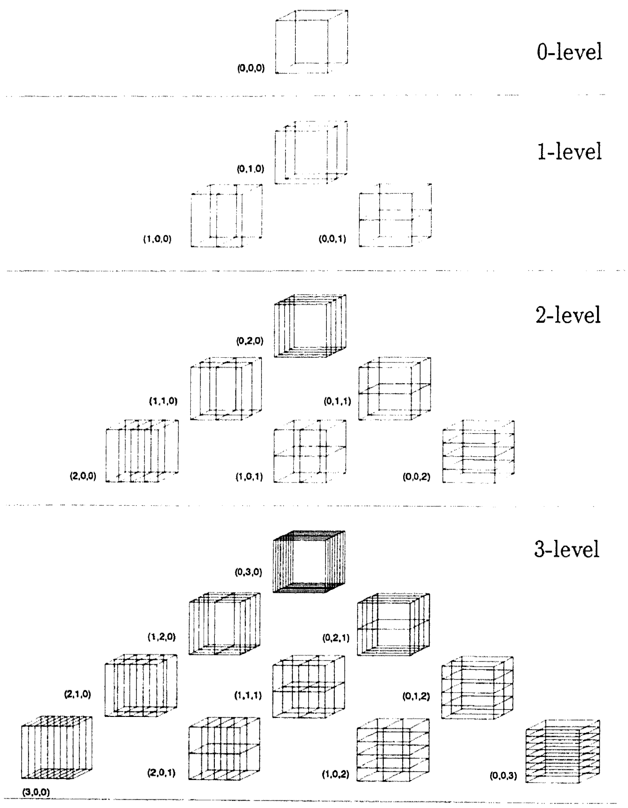


Figure 1: An impression of the grids $\mathcal{R}_{l,m,n}$ with $l, m, n \geq 0$.
In this figure a cell on the basic grid $\mathcal{R}_{0,0,0}$ is shown, together with its refinements on the grids $\mathcal{R}_{l,m,n}$, $l + m + n = \ell$, on all the ℓ-levels, $\ell = 1, 2, 3$.

Let the discrete equations, that model the PDE, be defined on a computational domain Ω. We assume that the computational domain Ω, an open set in $\mathbb{R}^3$, is not infinite, but that it consists of only a finite number of cells in the basic grid $\mathcal{R}_{0,0,0}$. Without loss of generality we may assume that the coordinates of all points in the closure $\overline{\Omega}$ of Ω are non-negative.

For all grids we introduce the notion of nodal point (=cell vertex), cell, cell center, cell face or cell edge with their obvious meaning:

- a *nodal point* or *cell vertex* is an element of $\mathcal{R}_{l,m,n}$:

$$\mathcal{V}_{i,j,k} = \mathcal{V}_{i,j,k,l,m,n} = (i2^{-l}, j2^{-m}, k2^{-n}) \ .$$

- a *cell* is the interior of an elementary cube in the grid:

$$\begin{aligned} \mathcal{C}_{i,j,k} &= \mathcal{C}_{i,j,k,l,m,n} \\ &= \{(x,y,z) \ ; \quad |(i+\tfrac{1}{2})2^{-l} - x| < 2^{-l-1} \ , \\ &\qquad\qquad\qquad\ |(j+\tfrac{1}{2})2^{-m} - y| < 2^{-m-1} \ , \\ &\qquad\qquad\qquad\ |(k+\tfrac{1}{2})2^{-n} - z| < 2^{-n-1} \quad \} \ . \end{aligned}$$

  We notice that a cell is an open set in $\mathbb{R}^3$.

- a *cell center* is the center of gravity of a cell

$$\mathcal{E}_{i+\frac{1}{2},j+\frac{1}{2},k+\frac{1}{2}} = ((i+\tfrac{1}{2})2^{-l}, (j+\tfrac{1}{2})2^{-m}, (k+\tfrac{1}{2})2^{-n}) \ .$$

- a *cell edge* is an open ended segment between two neighbouring nodal points. We distinguish 3 types of edges:

  - an x-edge

$$\mathcal{E}_{i+\frac{1}{2},j,k} = \left\{(x, j2^{-m}, k2^{-n});\ |(i+\tfrac{1}{2})2^{-l} - x| < 2^{-l-1}\right\},$$

  - an y-edge

$$\mathcal{E}_{i,j+\frac{1}{2},k} = \left\{(\ i2^{-l}, y, k2^{-n});\ |(j+\tfrac{1}{2})2^{-m} - y| < 2^{-m-1}\right\},$$

  - a z-edge

$$\mathcal{E}_{i,j,k+\frac{1}{2}} = \left\{(i2^{-l}, j2^{-m}, z);\ |(k+\tfrac{1}{2})2^{-n} - z| < 2^{-n-1}\right\}.$$

- a *cell face* is an open rectangle between two neighbouring cells. We distinguish 3 types of cell faces:

  - an x-face

$$\mathcal{E}_{i,j+\frac{1}{2},k+\frac{1}{2}} =$$

$$\left\{(i2^{-l}, y, z);\ |(j+\tfrac{1}{2})2^{-m} - y| < 2^{-m-1},\ |(k+\tfrac{1}{2})2^{-n} - z| < 2^{-n-1}\right\} \ ,$$

  - an y-face

$$\mathcal{E}_{i+\frac{1}{2},j,k+\frac{1}{2}} =$$

$$\left\{(x, j2^{-m}, z);\ |(i+\tfrac{1}{2})2^{-l} - x| < 2^{-l-1},\ |(k+\tfrac{1}{2})2^{-n} - z| < 2^{-n-1}\right\} \ ,$$

  - a z-face

$$\mathcal{E}_{i+\frac{1}{2},j+\frac{1}{2},k} =$$

$$\left\{(x, y, k2^{-n});\ |(i+\tfrac{1}{2})2^{-l} - x| < 2^{-l-1},\ |(j+\tfrac{1}{2})2^{-m} - y| < 2^{-m-1}\right\} \ .$$

We call the cell interiors, cell faces, cell edges and cell vertices the *cell elements* in the grid.

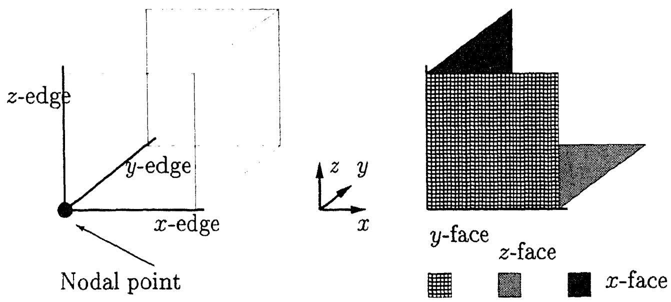


Figure 2: A patch: the nodal point, the cell edges and cell faces.

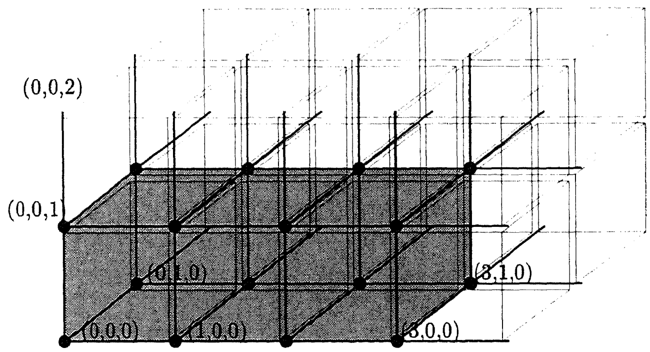


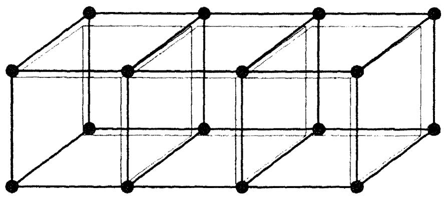
Figure 3: Structure of the patches making a connected block $[0,3] \times [0,1] \times [0,1]$.

1.2 Patches

The above considerations were independent of the realisation in a data structure. We aim at the construction of a data structure for adaptive computations. This implies that we are interested in all the possible grids $\mathcal{R}_{l,m,n}$, with $l,m,n \geq 0$, as far as they cover the domain $\overline{\Omega}$. However, a priori we do not know what grids and what parts (what cell elements) of these grids will be needed in a computation.

Therefore we realise a data structure in which all cell elements that cover $\overline{\Omega}$ on the basic grid $\mathcal{R}_{0,0,0}$ will be represented. Further, all cells on the grid $\mathcal{R}_{l,m,n}$, $l,m,n \geq 0$, may exist in the data structure, provided that there exist also cells that cover the same space in the coarser grids $\mathcal{R}_{l-1,m,n}$, $\mathcal{R}_{l,m-1,n}$ and $\mathcal{R}_{l,m,n-1}$. Notice that, for each of these 3 grids these *father cells* are uniquely determined. However, if any of the indices $l-1$, $m-1$ or $n-1$, is negative, we do *not* require corresponding father cells to exist in the coarser grid. We notice that in all aspects the data structure is (and remains) symmetric with respect to the three coordinate directions.

As $\overline{\Omega}$ is the union of the closure of a finite number of cells in $\mathcal{R}_{0,0,0}$, we see that for all $l,m,n \geq 0$ the closed domain $\overline{\Omega}$ is exactly the union of a finite number of (open) cells, cell faces, cell edges and cell vertices! We also see that the number of cell vertices is larger than the number of cells (because $\overline{\Omega}$ has boundaries on all sides), but asymptotically for fine grids or large domains the difference between these numbers becomes less significant.



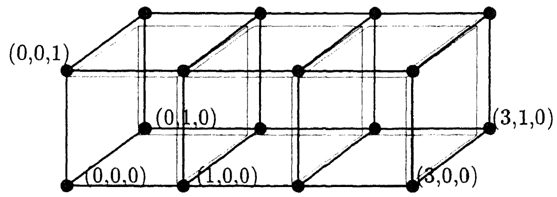


Figure 4: All 16 patches in the block $[0,3] \times [0,1] \times [0,1]$, unused parts are not shown.

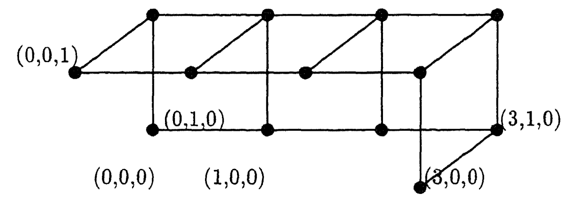


Figure 5: The 13 thin patches in the block $[0,3] \times [0,1] \times [0,1]$.

Asymptotically for fine grids, if the number of cells is $\mathcal{O}(N)$, also the number of cell vertices is $\mathcal{O}(N)$, and the number of cell faces and cell edges is $\mathcal{O}(3N)$.

Because we want to be able to handle all possible cell elements in the data structure for adaptive computations, and because we want to minimise the number of pointers necessary, we introduce the 'patch' as an elementary unit in the data structure.

With each nodal point $\mathcal{V}_{i,j,k}$ in the grid $\mathcal{R}_{l,m,n}$ we associate the *patch*, i.e. the union of a vertex, 3 edges, 3 faces and the corresponding cell interior in the positive direction: $\mathcal{P}_{i,j,k} = \mathcal{P}_{i,j,k,l,m,n}$,

$$\begin{aligned}\mathcal{P}_{i,j,k} = \mathcal{V}_{i,j,k} \;&\cup \mathcal{E}_{i+\frac{1}{2},j,k} \cup \mathcal{E}_{i,j+\frac{1}{2},k} \cup \mathcal{E}_{i,j,k+\frac{1}{2}} \\ &\cup \mathcal{E}_{i,j+\frac{1}{2},k+\frac{1}{2}} \cup \mathcal{E}_{i+\frac{1}{2},j,k+\frac{1}{2}} \cup \mathcal{E}_{i+\frac{1}{2},j+\frac{1}{2},k} \cup \mathcal{C}_{i,j,k} \,.\end{aligned} \tag{3}$$

This implies that for each grid the domain Ω is fully covered by a union of patches, whereas the domain $\overline{\Omega}$ is covered exactly by the same union of patches, with the exception of half of the boundary of $\overline{\Omega}$. In order to complete the covering of $\overline{\Omega}$ we have to add

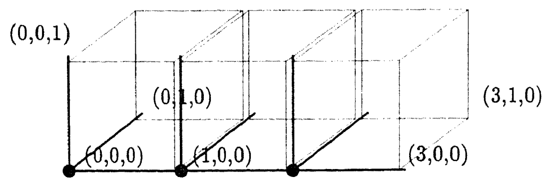


Figure 6: The 3 complete patches in the block $[0,3] \times [0,1] \times [0,1]$.

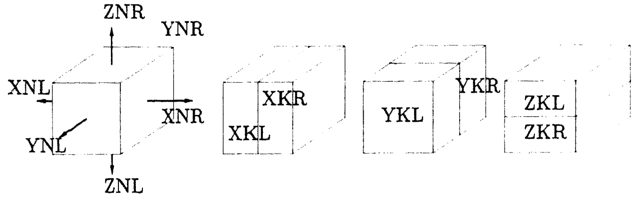


Figure 7: The 6 neighbours and 6 kids of a cell.

partial patches, so called *thin patches* at the right-hand side of the domain (i.e. at the side of the positive coordinate directions).

These thin patches only account for a vertex and possibly also edges and faces. They don't have a corresponding volume.

The *thin patches* and the *complete patches* (i.e. the patches that represent a volume $\mathcal{C}_{i,j,k}$) are the basic elements of the data structure that is described in this report later. Each patch is identified by the level and the coordinates of its vertex. Later we shall also see that it can also be identified by its unique *patch number.*

It will be clear that a patch representing a volume $\mathcal{C}_{i,j,k}$, automatically also represents all faces, edges and the vertex at the left-hand side of the volume. A patch $\mathcal{P}_{i,j,k}$ in any case represents the vertex $\mathcal{V}_{i,j,k}$. However, if the volume is not represented (is not present), a choice of faces and edges can be represented by the patch. Thus, different types of thin patches are possible (see Figure 5). The book-keeping of what faces and/or edges are represented by a particular the patch is stored in the data structure as a 'plain patch property'. These patch properties are listed in Table 1.

Table 1: The plain properties of a patch.

for patch $\mathcal{P}_{i,j,k}$	
property code	meaning
Complete	$\mathcal{C}_{i,j,k} \subset \mathcal{P}_{i,j,k}$
XWall	$\mathcal{E}_{i,j+\frac{1}{2},k+\frac{1}{2}} \subset \mathcal{P}_{i,j,k}$
YWall	$\mathcal{E}_{i+\frac{1}{2},j,k+\frac{1}{2}} \subset \mathcal{P}_{i,j,k}$
ZWall	$\mathcal{E}_{i+\frac{1}{2},j+\frac{1}{2},k} \subset \mathcal{P}_{i,j,k}$
XEdge	$\mathcal{E}_{i+\frac{1}{2},j,k} \subset \mathcal{P}_{i,j,k}$
YEdge	$\mathcal{E}_{i,j+\frac{1}{2},k} \subset \mathcal{P}_{i,j,k}$
ZEdge	$\mathcal{E}_{i,j,k+\frac{1}{2}} \subset \mathcal{P}_{i,j,k}$

Table 2: The neighbours of a patch.

relative patch $\mathcal{P}_{i,j,k}$		
relation	code	meaning
left x-neighbour	XNL	$\mathcal{P}_{i-1,j,k}$
right x-neighbour	XNR	$\mathcal{P}_{i+1,j,k}$
left y-neighbour	YNL	$\mathcal{P}_{i,j-1,k}$
right y-neighbour	YNR	$\mathcal{P}_{i,j+1,k}$
left z-neighbour	ZNL	$\mathcal{P}_{i,j,k-1}$
right z-neighbour	ZNR	$\mathcal{P}_{i,j,k+1}$

Table 3: The kids of a patch.

for patch $\mathcal{P}_{i,j,k,l,m,n}$ the following kids may exist		
relation	code	meaning
left x-kid	XKL	$\mathcal{P}_{2i,j,k,l+1,m,n}$
right x-kid	XKR	$\mathcal{P}_{2i+1,j,k,l+1,m,n}$
left y-kid	YKL	$\mathcal{P}_{i,2j,k,l,m+1,n}$
right y-kid	YKR	$\mathcal{P}_{i,2j+1,k,l,m+1,n}$
left z-kid	ZKL	$\mathcal{P}_{i,j,2k,l,m,n+1}$
right z-kid	ZKR	$\mathcal{P}_{i,j,2k+1,l,m,n+1}$

1.3 Neighbours and kids

Because all grids have a regular rectangular structure, each cell or patch in a grid has at most six direct *neighbours*. For the patch $\mathcal{P}_{i,j,k}$ these are identified as the left- and right-, x-, y- or z- neighbour. This is summarised in Table 2.

We call a cell (or patch) a *kid* of another cell (or patch), which is called the *father*, if it represents a subset of the father and if it is obtained from the father by a single step of refinement (one level difference). Similarly we introduce the notion of *grandfather* and *grandson*. Since an elementary cube in the $\mathcal{R}_{l,m,n}$-grid has dimension 2^{-l}, 2^{-m} and 2^{-n} in the x-, y- and z-direction respectively, the volume of each *father cell* is double the volume of the *kid cell* and each cell has at most six possible kids. Such kids can be found by splitting the father cell in two equal blocks. This can be done in the x-, y- and z-direction. Thus we can identify the left- and right-, x-, y- or z- kid of a patch. This is summarised in Table 3.

We extend the requirement for a kid cell (at the beginning of Section 1.2) that each cell on a mesh $\mathcal{R}_{l,m,n}$ should have a father in the mesh $\mathcal{R}_{l-1,m,n}$, $\mathcal{R}_{l,m-1,n}$ and $\mathcal{R}_{l,m,n-1}$ (unless a corresponding index becomes negative) to a similar requirement for patches. Thus, we require that for $l > 0$ any patch on $\mathcal{R}_{l,m,n}$ should have a corresponding father in the $\mathcal{R}_{l-1,m,n}$-mesh. This father is called the x-father. Similar relations should hold for the y- and z-direction. This means that, if a patch is refined in some direction, then the patch should have a father in that particular direction. This is summarised in Table 4.

1.4 Ghost patches, the root patch

We already noticed that the patches are the elementary unit in our data structure. They are identified by their patch numbers (or by their level l, m, n and integer coordinates

Table 4: The fathers of a patch.

for patch $\mathcal{P}_{i,j,k,l,m,n}$ the following fathers should exist:			
relation	code	meaning	only if
x-father	XF	$\mathcal{P}_{\lfloor i/2 \rfloor,j,k,l-1,m,n}$	$l > 0$
y-father	YF	$\mathcal{P}_{i,\lfloor j/2 \rfloor,k,l,m-1,n}$	$m > 0$
x-father	XF	$\mathcal{P}_{i,j,\lfloor k/2 \rfloor,l,m,n-1}$	$n > 0$

i, j, k) and they are linked to each other by *neighbour relations* with patches that belong to the same grid, and by *father-kid relations* between the different levels of refinement. Until now, all patches were related to a grid $\mathcal{R}_{l,m,n}$, $l, m, n \geq 0$. We extend this in this section to some levels with $l, m, n < 0$.

By the requirement that for $l > 0$ any patch on $\mathcal{R}_{l,m,n}$ should have a corresponding father in the $\mathcal{R}_{l-1,m,n}$-mesh, and similar relations for the y- and z-direction, a straightforward 3-fold intertwined bin-tree structure is created between all patches of the structure. The roots of this tree structure are the patches on the $\mathcal{R}_{0,0,0}$-grid.

In order to generate a unique root, instead of the multiple roots on the $(0, 0, 0)$-level, we extend the data structure for grids $\mathcal{R}_{l,m,n}$, for which $l, m, n \leq 0$, as follows. For all patches in the 0-level $\mathcal{R}_{l,m,n}$ $(l, m, n = 0)$ we construct their x-, y- and z- father on the level $l+m+n = -1$. Recursively we construct all fathers on levels with smaller $l+m+n$. Because the domain $\overline{\Omega}$ is finite, this process results eventually in one unique patch from which all the patches on the $\mathcal{R}_{0,0,0}$-grid are descendants. The level of this unique *root patch* is simply identified as $(l_{\text{root}}, m_{\text{root}}, n_{\text{root}})$, i.e. the *root level*, with

$$\begin{aligned} l_{\text{root}} &= -\lceil \, {}^2\log|\Omega_x| + 1 \, \rceil \,, \\ m_{\text{root}} &= -\lceil \, {}^2\log|\Omega_y| + 1 \, \rceil \,, \\ n_{\text{root}} &= -\lceil \, {}^2\log|\Omega_z| + 1 \, \rceil \,, \end{aligned}$$

where $|\Omega_d|$ is the length of the block $\overline{\Omega}$ in the d-direction, $d = x, y, z$. Without loss of generality the root patch can be identified as $\mathcal{P}_{0,0,0,l_{\text{root}},m_{\text{root}},n_{\text{root}}}$.

This construction of the unique root patch makes it possible to reach a patch from the basic grid $\mathcal{R}_{0,0,0}$ starting from the root patch, in the same way as it is possible to reach an arbitrary patch in the structure from the corresponding patch on the basic level by stepping up in the tree. This implies that now all patches can be easily reached, starting from the root patch.

Thus, all patches in the structure are related to each other by father-kid relations in a simple 3-fold intertwined bin-tree structure with a unique root patch. All patches on negative levels are introduced to create a consistent data structure, and we don't associate them with a geometric meaning. These patches with a negative level are called *ghost patches*.

1.5 Local refinements, boundaries

As mentioned before, on the basic level all the domain $\overline{\Omega}$ is covered by patches: complete patches and additional thin patches on the right-hand-side boundary. On a refined level $(l + m + n > 0)$ the patches do *not* necessarily cover all the domain $\overline{\Omega}$.

A *local refinement* on a grid $\mathcal{R}_{l,m,n}$ consists of the closure of the union of a number of cells in this grid $\mathcal{R}_{l,m,n}$. To represent the local refinement, for each cell $C_{i,j,k}$ we have

Table 5: Levels in the data structure.

for a patch $\mathcal{P}_{i,j,k,l,m,n}$ we distinguish the following level properties:	
l, m, n are either all non-negative, or l, m, n are all non-positive.	
If $l+m+n=0$	$\mathcal{P}_{i,j,k,l,m,n}$ is on the *basic* level
If $l+m+n<0$	$\mathcal{P}_{i,j,k,l,m,n}$ is on a *ghost* level
If $l+m+n>0$	$\mathcal{P}_{i,j,k,l,m,n}$ is on a *refined* level
$l \geq l_{\text{root}}$, $\quad m \geq m_{\text{root}}$, $\quad n \geq n_{\text{root}}$.	

Table 6: The boundary properties of a patch.

for patch $\mathcal{P}_{i,j,k}$	
property code	meaning
XBdyWall	$\mathcal{E}_{i,j+\frac{1}{2},k+\frac{1}{2}} \subset \partial\Omega$
YBdyWall	$\mathcal{E}_{i+\frac{1}{2},j,k+\frac{1}{2}} \subset \partial\Omega$
ZBdyWall	$\mathcal{E}_{i+\frac{1}{2},j+\frac{1}{2},k} \subset \partial\Omega$
XBdyEdge	$\mathcal{E}_{i+\frac{1}{2},j,k} \subset \partial\Omega$
YBdyEdge	$\mathcal{E}_{i,j+\frac{1}{2},k} \subset \partial\Omega$
ZBdyEdge	$\mathcal{E}_{i,j,k+\frac{1}{2}} \subset \partial\Omega$
BdyPoint	$\mathcal{E}_{i,j,k} \subset \partial\Omega$

a patch $\mathcal{P}_{i,j,k}$ in the data structure. In addition we need a number of thin patches (at right-hand-side boundary of the local refinement) in order to complete the closure of the domain. We notice that also these thin patches (should) have there corresponding fathers in the coarser grids.

Local refinements do not necessarily cover connected subdomains of $\overline{\Omega}$. the only requirement for the existence of a refinement is the existence of sufficient father cells (father patches) on (all) coarser grids.

We emphasise that local refinements, as the original domain $\overline{\Omega}$, are closed sets. They include their boundaries. The boundary $\partial\Omega$ of Ω is called the *domain boundary* or, briefly, the *boundary*. This boundary is certainly represented by the patches in the basic grid $\mathcal{R}_{0,0,0}$. The boundary can also be represented (in part) on the refined grids. Patches that contain part of the boundary are called *boundary patches.*

Different faces and/or edges of a patch can represent part of the boundary. It will be clear that, if a face belongs to $\partial\Omega$, then at least two edges belong also to $\partial\Omega$; and if an edge belongs to $\partial\Omega$, then also the vertex of the patch. In order to recognise which part of a patch belongs to the boundary we distinguish several *boundary properties.* These properties are summarised in Table 6.

On the refined levels, generally the local refinements will cover only part of the domain Ω. This implies that parts of the boundary of the local refinements will be in the interior of Ω. The boundary of a local refinement (as well as the boundary of Ω itself) is called a *green boundary* and patches that represent part of the green boundary are called *green patches.* This implies that all (sub)sets of $\overline{\Omega}$ that are represented at the basic or refined levels are bounded by a green boundary. It will be clear that the family of all boundary patches is a subset of the family of green patches.

Table 7: The green properties of a patch.

for patch $\mathcal{P}_{i,j,k}$	
property code	meaning
XGrnWall	$\mathcal{E}_{i,j+\frac{1}{2},k+\frac{1}{2}} \subset \partial\mathcal{G}$
YGrnWall	$\mathcal{E}_{i+\frac{1}{2},j,k+\frac{1}{2}} \subset \partial\mathcal{G}$
ZGrnWall	$\mathcal{E}_{i+\frac{1}{2},j+\frac{1}{2},k} \subset \partial\mathcal{G}$
XGrnEdge	$\mathcal{E}_{i+\frac{1}{2},j,k} \subset \partial\mathcal{G}$
YGrnEdge	$\mathcal{E}_{i,j+\frac{1}{2},k} \subset \partial\mathcal{G}$
ZGrnEdge	$\mathcal{E}_{i,j,k+\frac{1}{2}} \subset \partial\mathcal{G}$
GrnPoint	$\mathcal{E}_{i,j,k} \subset \partial\mathcal{G}$

As for the domain boundary, different faces and/or edges of a patch can represent part of a green boundary. Therefore, we also recognise different possible *green properties* for a patch. Let $\mathcal{G}_{l,m,n}$ be the subset of Ω that is covered by a local refinement on $\mathcal{R}_{l,m,n}$, then $\partial\mathcal{G} = \overline{\mathcal{G}}_{l,m,n}\backslash\mathcal{G}_{l,m,n}$ is the green boundary on level (l,m,n). Because of their location on a green boundary, for green patches we speak of: green faces, green edges and green points, in the same way as we speak of boundary faces, boundary edges and boundary points for the boundary patches. The different green properties are summarised in Table 7.

2 The data structure

Although PASCAL, C++ and other, modern languages are really well suited to properly implement a data structure as described in this report, for some practical reasons it has been decided that it should also be implemented in FORTRAN. This infers restrictions in the way a data structure can be implemented.

Our experience is that the construction of a FORTRAN implementation is enhanced if first a prototype is made available in a better equipped language. Therefore, a prototype of the essential parts of the data structure is built in PASCAL, taking into account the restrictions that are inherent to the use of FORTRAN. This implies that the useful features as pointers and recursive procedures, that are available e.g. in PASCAL but not in FORTRAN, were abandoned in the prototype.

The PASCAL prototype is found in Appendix section A.1. In Appendix section A.3 and A.4 the user interface for the FORTRAN implementation is given.

2.1 Patch numbers and pointers

As we have coordinates, properties and neighbour- or father-kid- relations in the data structure, with each patch we also associate *coordinates*, *pointers* and *properties*. Further we want to provide a patch with a set of real numbers for the representation of the numerical data: its *data contents*. These numerical data can be associated with the vertex, the cell etc., as a user likes it.

All data are kept in 3 large arrays: an integer array (PNTR), a Boolean array (PPTY) and a real array (DATA). These are all two-dimensional arrays with MNOP columns, where MNOP is the *maximum number of patches* that is allowed in an implementation. The p-th column of each of these arrays is associated with the p-th patch in the data structure. This number p, the *patch number* is a unique natural number, identifying this

Table 8: The row numbers ('code') for the integer coordinates.

index	meaning
XL	l
YL	m
ZL	n
XJ	i
YJ	j
ZJ	k

patch. However, the number p has no particular meaning and it may be changed during the computation, provided that all references to this patch (by means of this number) are changed correspondingly.

However, there are two exceptions: $p = 0$ and $p = 1$ are special patch numbers: $p = 0$ represents the *nil pointer* , a pointer which does **not** refer to a patch (or, refers to the non-existence of a patch), $p = 1$ refers to the unique *root patch.*

For a patch the references to other patches, the *pointers*, are all found in the integer array (PNTR), together with the *integer coordinates* l, m, n, i, j, k, which are also called the *indices* of the patch $\mathcal{P}_{i,j,k,l,m,n}$. Different row-numbers are associated with the different indices or different pointers. Row-numbers for the different indices (integer coordinates) of a patch are summarised in Table 8. Which row numbers are associated with the particular pointers to neighbours, fathers and kids are found as 'code' in the Tables 2, 4, 3. The 'code' represents a unique integer that serves as the row index for the array PNTR.

Thus, the pointers of a patch are implemented as integers in the array PNTR, referring to the corresponding (father-, neighbour- or kid-) patch number. Properties of patches are implemented similarly as Booleans in the array PPTY.

2.2 Pointers and coordinates: the array PNTR

The integer array 'PNTR', at least dimensioned [FstPtr:LstPtr, 0:MNOP], is used for keeping the pointers. For each patch a set of 15 pointers is provided: XF, YF, ZF, XKL, XKR, YKL, YKR, ZKL, ZKR, XNL, XNR, YNL, YNR, ZNL, ZNR. The meaning of these pointers is found in the Tables 2, 3 and 4.

Another part of the same integer array PNTR, dimensioned as [FstIdx:LstIdx, 0:MNOP], is used for keeping the integer coordinates (i.e. the indices): XL,YL,ZL, XJ,YJ,ZJ. E.g. the element PNTR[YNL,p] contains the patch-number of the left-hand y-neighbour of the patch p (i.e. the patch with patch-number p) and for the same patch $\mathcal{P}_{i,j,k,l,m,n}$ the index k, is found in PNTR[ZJ,p] and the index l in PNTR[XL,p].

Remark:

Because the order of the row-indices has no intrinsic meaning, for the implementation we use named integer constants (the 'code') to identify the row-number. For the array PNTR, the names FstIdx, LstIdx, FstPtr (=LastIdx+1) and LstPtr are introduced to indicate the first (last) pointer or integer coordinate present. These names facilitate loops over row-elements.

In the actual implementation we take FstIdx=1, LstIdx=6, FstPtr=7, LstPtr=21.

We dimension the array PNTR as [FstIdx:LstPtr, 0:MNOP]. The elements of PNTR[*,0], the *nil patch*, are all initialised (and kept) equal to zero. That makes that any pointer from the nil patch is again the nilpointer. This simplifies the implementation because it makes that "the kid or neighbour of a not-existing patch doesn't exist".

Remark:
During the computation we may expect that the maximum number of patches (MNOP) is never used. The active patches present in the data structure are found for the patch numbers $1 \leq p \leq$ NOP (*number of patches*). However, the integer variable NOP does not (always) represent the number of patches that is active in the data structure, because – previously living– patches may have disappeared from the data structure and still keep a patch number. These empty patch numbers for *dead patches* can be re-used (for another patch) in the sequel of a computation.

2.3 Properties: the array PPTY

Various properties of a patch are stored in the Boolean array: PPTY. Many of these properties (e.g. whether the patch is located near a boundary) can be derived from the pointer structure, but to avoid many trivial recomputations, some properties can better be stored independently in the data structure. (A routine CHKPPT can be made available to check the consistency of the data.) Also some other information in the form of properties, is used in the data structure and additional properties, *flags*, can be defined by the user.

Properties of a *patch* that are of interest are found in the Tables 1, 6 and 7. These tables give also the row-number (the 'code') where this information can be found in the array. Other properties (flags) given to a patch are 'PregnantX', 'PregnantY' and 'PregnantZ', which indicate that the cell should be refined in the x-, y- or z-direction, at the first possible occasion. The flag 'Sentenced' denotes that the cell should be removed at the first possible occasion and the flag 'Dead' denotes that the patch has been removed from the data structure (the patch number is free for re-use).

Remark:
While the order of the Boolean properties in the array has no intrinsic meaning, the actual implementation is made by named integer constants. The first property is also called 'FstPpt' and the last property 'LstPpt'. This means that the array PPTY should be dimensioned at least as [FstPpt:LstPpt,0:MNOP]. The availability of 'FstPpt' and 'LstPpt' also enables the construction of loops over all row-elements of the array.

2.4 Data contents: the array DATA

The data contents of the data structure is contained in the real array DATA dimensioned as [1:MNOD,1:MNOP], where MNOD is the maximum number of real data that can be stored per patch. The choice of MNOD depends on the application and is left to the user, as is the distribution of the data over the different rows. In fact, the basic procedures for handling the data structure do not refer to the array DATA.

Remark:
In addition to –or instead of– the array DATA, the user can introduce his own arrays as

(additional) storage, possibly of different data type, for any data he needs in the structure. The only condition is that, in one of its indices, it should should be dimensioned '1:MNOP'.

3 The actions on the data structure

3.1 Construction of the data structure

For the construction and the handling of the data structure, the following routines are available to the user. Notice that routines are available both for creating parts of the data structure, as well as removing parts of it. The space that is made free by the removal of some parts will be used again by the newly generated parts.

It is useful to know that the active part of the data structure is always the closure of a set of open cells. These cells are created and removed. The routines mentioned below keep the data structure up-to-date. They take care of all actions necessary to represent the closure of all active cells (they take care of green boundaries etc.).

Many routines get access to a cell through the pointer p of the corresponding patch. This is the usual way to know a patch. If necessary, a routine 'GetPtr' can be used to obtain this patch number p if the integer coordinates i, j, k, l, m and n are given.

- **routine** InizBasis3. This routine, which has no parameters, should be called once before the data structure is used. This routine (re-) initialises the data structure.

- **routine** MakeBlock (*dimx*,*dimy*,*dimz*, *xcycl*,*ycycl*,*zcycl*). This routine has 3 integer parameters: *dimx*, *dimy* and *dimz*, and 3 Boolean parameters *xcycl*, *ycycl* and *zcycl*. This routine creates the data structure for the domain $\overline{\Omega}$, where Ω is (topologically equivalent with) a rectangular parallelepiped on the basic grid $\mathcal{R}_{0,0,0}$. A call of MakeBlock will result in active cells on the 0-level in a parallelepiped $\Omega = (0, dimx) \times (0, dimy) \times (0, dimz)$.

  Remark:
  To create a domain Ω with a different shape, first an enclosing block should be created on the basic grid. Then, before further operations on the data structure are made, the superfluous cells $\mathcal{C}_{i,j,k,0,0,0}$ should be removed from the enclosing block by the routine RemoveCell.

  The parameters *dcycl* ($d = x, y, z$) denote the topological structure of the block. To create a true topological block, all parameters *dcycl* should be set **false**. If *dcycl* is **true**, then the domain will be periodic in the d-direction: then on the 0-level the cells with d-coordinate 0 are identified with the corresponding cells with the d-coordinate *dimd*. Then the cells with the centre at d-coordinate $\frac{1}{2}$ are the right-neighbours of the corresponding cells with the centre at d-coordinate $dimd - \frac{1}{2}$; similarly the left-neighbours of the cells with the centre at d-coordinate $\frac{1}{2}$ are cells with the centre at d-coordinate $dimd - \frac{1}{2}$. In this way e.g. periodic boundary conditions are easily realised and also 1- and 2-dimensional problems can easily be simulated.

- **routine** MakeFamily (l,m,n, i,j,k). This routine has 6 integer parameters: the integer coordinates of a cell that should become active in the structure. If possible, this routine adds cell $C_{i,j,k,l,m,n}$ to the active part of the data structure, together with all its necessary ancestors. It is possible to add the cell if it doesn't already exist and if $C_{i,j,k,l,m,n} \subset \Omega$.

- **routine** MakeOffspring (p). This routine has one pointer-parameter p. For patch p this routine adds to the active structure all its kid-cells for which the creation is possible. It is possible to create the (not already existing) kids in the d-direction ($d = x, y, z$) only if the *Pregnantd* flag is **true** for patch p and if sufficient fathers exist for the new cell.

- **routine** MakeCell (l,m,n,i,j,k, p). This routine has 6 integer parameters and one pointer-parameter p. If possible, this routine adds the cell $C_{i,j,k,l,m,n}$ to the active part of the data structure. It is possible to add a cell if it doesn't exist already and if all its fathers (on non-negative levels) exist.

  A user can always set the nil-pointer for p. (A sophisticated user may save some computing time by setting p equal to the patch number of $\mathcal{P}_{i,j,k,l,m,n}$).

- **routine** RemoveOffspring (p). This routine has one pointer-parameter p. This routine removes all kid-cells for the patch p, provided that this removal is possible. The removal of a cell is possible if that cell is marked as 'Sentenced' and if it has no kids.

- **routine** RemoveCell (p). This routine has one pointer-parameter p. If possible, this routine removes the cell in patch p from the active structure. The removal is possible if the cell has no kids.

- **routine** GetPtr (l,m,n,i,j,k). This routine has 6 integer parameters and it delivers a pointer. The routine delivers the patch number (pointer) for $\mathcal{P}_{i,j,k,l,m,n}$.

3.2 Scanning the patches

All computations on the data structure are performed by scanning all necessary patches and doing the computations for each patch when it is visited. Therefore, it is basic to be able to visit a selected set of patches in a particular order and to perform an action during that visit.

Below a routine is made available to visit the patches of a particular grid. Then it is easy to create a routine that can perform an action on the patches from (part of) all grids in a level. Further selection, within a grid, can be made part of the action performed.

The order in which all patches are visited is lexicographical (lexicographical in a specified permutation of the indices i, j, k). However, the order can be either in the forward or in the backward direction. Further, for the data structure there is no preference in coordinate direction. This leaves us with 48 possibilities for scanning all patches in a grid (any permutation of the coordinate directions can be chosen: x, y and z, and for any coordinate direction there is the freedom of back- or forward scanning, i.e. 8 possibilities for each permutation).

The choice between these orderings is made by a small input array 'myorder'. This is an integer array dimensioned [1:3]. To specify the ordering, the array should contain 3 integers corresponding to the codes from Table 3, such that one code is chosen for each coordinate direction (e.g. either YKL or YKR). The 3 resulting codes are placed in 'myorder' in any order. The codes denote from what side the scanning is started (YKL means: in the y-direction we start from the left), and their ordering in the array denotes their ordering in the lexicographical treatment.

E.g., the array (YKL,XKR,ZKL) corresponds with 3 nested loops: scanning over the y-direction in the outermost loop and scanning over the z-direction in the innermost loop; the y-scanning is from left to right, the x-scanning from right to left and the z-scanning from left to right.

For scanning the patches in this way, the following routine is available:

- **routine** ScanGrid (l,m,n, *myorder*, *DoPatch*). This routine has 3 integer parameters to specify the grid and a parameter-array 'myorder' to specify the order by which the scanning should take place. The last parameter *DoPatch* is the reference to a routine that specifies what action has to be performed in each patch that is visited.

  - This routine 'DoPatch (p)' has one input parameter, viz. the pointer p that indicates which patch is being visited. This routine should be provided by the user to specify what action is wanted.

  The routine ScanGrid scans all active patches in the set

$$\{\mathcal{P}_{i,j,k,l,m,n};\, i\in\mathbb{Z},\, j\in\mathbb{Z},\, k\in\mathbb{Z}\}$$

  in the order as specified by *myorder*. When a patch is visited, a call to the routine *DoPatch* is made.

A Appendix: the implementation

A.1 The PASCAL prototype

In this section we give a full description of the data structure in PASCAL.

```
   { -- starting point for a FORTRAN 77 code -- }
   { -- created: 13 August 1992            -- }
   { -- last corrections:  1992-11-16      -- }
   { -- author: P.W. Hemker                -- }

{ -- first a number of parameters for the include file -- }
const
{- MNOP  = 320000; -} { -- MaxNumberOfPatches   }
{- MNOD  =      4; -} { -- MaxNumberOfData      }
   MNOL  =     30;   { -- MaxNumberOfLevels    }
   LNOL  =    -12;   { -- LowestNumberOfLevels }
   nihil =      0;   { -- the nil pointer      }
   nowhere=  -999;   { -- no place in structure }

   { --  DIRECTIONS -- }
   { -- X = 1; Y = 2; Z = 3;     -- coordinate directions      -- }

   { --  INDICES -- }
   XL =  1; YL =  2; ZL =  3; { -- level indices              -- }
   XJ =  4; YJ =  5; ZJ =  6; { -- coordinate indices         -- }

   FstIdx = XL;  LstIdx = ZJ;

   { --  POINTERS -- }
   XF =  7; YF =  8; ZF =  9; { -- pointers to fathers        -- }

                              { -- pointers to kids           -- }
   XKL= 10; YKL= 11; ZKL= 12; { -- the kids (even coordinate) -- }
   XKR= 13; YKR= 14; ZKR= 15; { -- the kids (odd  coordinate) -- }

   FstKid =XKL; LstKid = ZKR;

                              { -- pointers to neighbours     -- }
   XNL= 16; YNL= 17; ZNL= 18; { -- in the negative direction  -- }
   XNR= 21; YNR= 20; ZNR= 19; { -- in the positive direction  -- }

   FstNgb   = XNL;  LstNgb = XNR;
   FstPtr   =  XF;  LstPtr = XNR;

   { --  PROPERTIES -- }
   { -- plain properties -- }
   Complete  =  1;
   XWall     =  2; YWall     =  3; ZWall     =  4;
   XEdge     =  5; YEdge     =  6; ZEdge     =  7;

   { -- boundary properties -- }
   XBdyWall  =  8; YBdyWall =  9; ZBdyWall = 10;
   XBdyEdge  = 11; YBdyEdge = 12; ZBdyEdge = 13;
   BdyPoint  = 14; { -- BdyCell = 15; -- }

   { -- green properties -- }
   XGrnWall  = 16; YGrnWall = 17; ZGrnWall = 18;
```

```
   XGrnEdge  = 19; YGrnEdge = 20; ZGrnEdge = 21;
   GrnPoint  = 22; GrnCell  = 23;

   { -- other properties -- }
   PregnantX = 24; PregnantY = 25; PregnantZ = 26;
   Sentenced = 27; Dead      = 28;

   FstBdyPpt = XBdyWall; LstBdyPpt = BdyPoint;
   GrnShift  = GrnPoint - BdyPoint;
   FstPpt    = Complete; LstPpt    = Dead;
type
   pointer   = integer;
   KidType   = FstKid..LstKid;
   NgbType   = FstNgb..LstNgb;
   order     = array[1.. 3] of KidType;
   strng     = array[1..24] of char;

{ -- Global variables, suitable for a COMMON -- }

var
   GeometryOK,
   XCycl, YCycl, ZCycl     :boolean;
   XSize, YSize, ZSize,
   XRoot, YRoot, ZRoot,
   RootPointer {= 1},
   LastSpace, NumberOfPatches :integer;
   TwoPow  :array [0..MNOL] of integer;
   NormalOrder                :order;

   PNTR      :array [FstIdx..LstPtr,0..MNOP] of integer;
   PPTY      :array [FstPpt..LstPpt,0..MNOP] of boolean;
   DATA      :array [     1..  MNOD,0..MNOP] of real;

   { -- Survey of the data structure (BASIS3) routines              -- }
   { -- The success of a data structure is its simplicity !         -- }
   { --                                                             -- }
   { -- error   (pointer,string,int)                                -- }
   { -- warning (pointer,string,int)                                -- }
   { --                                                             -- }
   { -- InizBasis3                                                  -- }
   { --                                                             -- }
   { -- GetPtr (n,m,l, i,j,k :int)pointer;                          -- }
   { -- GetLocation (pointer, var n,m,l, i,j,k :int);               -- }
   { -- GetKidType (pointer; var xtyp,ytyp,ztyp :KidType);          -- }
   { --                                                             -- }
   { -- SetPlnProperties (pointer);                                 -- }
   { -- SetGrnProperties (pointer);                                 -- }
   { -- SetBdyProperties (pointer);                                 -- }
   { --                                                             -- }
   { -- MakePatch(n,m,l, i,j,k :int; pointer)pointer;               -- }
   { --      -- takes care for all the pointers                     -- }
   { -- MakeCell(n,m,l, i,j,k :int; pointer);                       -- }
   { --      -- takes care of all properties                        -- }
   { -- MakeKid(KidType; pointer);                                  -- }
   { --      -- a cell is a complete patch with sufficient neighbours - }
   { -- MakeOffspring(pointer);                                     -- }
   { --      -- all kids of a father                                -- }
   { -- MakeFamily(n,m,l, i,j,k :int);                              -- }
```

```
   { --                                                           -- }
   { -- RemovePatch    (pointer);                                 -- }
   { -- RemoveCell     (pointer);                                 -- }
   { -- RemoveOffspring(pointer);                                 -- }
   { --                                                           -- }
   { -- ScanGrid (n,m,l :int; order; DoPatch )                    -- }
   { -- ScanLevel(lev, xmin,xmax,ymin,ymax,zmin,zmax :int;        -- }
   { --            order; DoGrid )                                -- }
   { --                                                           -- }
   { -- MakeBlock (n,m,l, xcyclic,ycyclic,zcyclic :boolean);      -- }
   { --                                                           -- }
   { -- ShowPatch (pointer);                                      -- }
   { -- ShowGrid  (n,m,l:int);                                    -- }
   { -- ShowLevel (lev:int);                                      -- }
   { -- ShowAll;                                                  -- }
   { -- DumpAll (name :strng);                                    -- }
   { --                                                           -- }


procedure GetLocation (p :pointer; var n,m,l, i,j,k :integer); forward;
procedure ShowPatch (patch: pointer); forward;
procedure ShowAll; forward;


procedure error (p :pointer; name: strng; nr:integer);
begin { -- hard error message -- }
      writeln('fatal error in routine ',name);
      ShowPatch(p);
      writeln('error number = ',nr:0);
end;

procedure warning (p :pointer; name: strng; nr:integer);
begin { -- soft error message -- }
      writeln('message from routine ',name);
      ShowPatch(p);
      writeln('message number = ',nr:0);
end;



procedure InizBasis3 ;
var  { -- initialisation of the data structure }
   i :integer;
begin
   { -- makes a pointer of the nil pointer the nil pointer! }
   for i:= FstIdx to LstIdx do PNTR[i,nihil]:= nowhere;
   for i:= FstPtr to LstPtr do PNTR[i,nihil]:= 0;
   for i:= FstPpt to LstPpt do PPTY[i,nihil]:= false;
   for i:=      1 to   MNOD do DATA[i,nihil]:= 0.0;

   TwoPow[0]:= 1; for i:= 1 to MNOL do TwoPow[ i]:= TwoPow[i-1]*2;
   GeometryOK:= false;
   NormalOrder[1]:= XKL;
   NormalOrder[2]:= YKL;
   NormalOrder[3]:= ZKL;
   NumberOfPatches:= 0; LastSpace:= 1;
end;
```

```
function GetPtr (n,m,l, i,j,k :integer):pointer;
var
   p :pointer;
   t, nn,mm,ll :integer;
begin
 if XCycl then begin if n<0 then else i:= i mod (XSize*TwoPow[n]) end;
 if YCycl then begin if m<0 then else j:= j mod (YSize*TwoPow[m]) end;
 if ZCycl then begin if l<0 then else k:= k mod (ZSize*TwoPow[l]) end;

 if (n<XRoot) or (m<YRoot) or (l<ZRoot) then GetPtr:= nihil else
 if (i<0) or (j<0) or (k<0) then GetPtr:= nihil else
 if (trunc(i/TwoPow[n-XRoot])<>0) or (trunc(j/TwoPow[m-YRoot])<>0) or
    (trunc(k/TwoPow[l-ZRoot])<>0) then GetPtr:= nihil else
 begin
   p:= 1; { -- i.e. RootPointer -- }
   if n<0 then nn:=n else nn:= 0;
   if m<0 then mm:=m else mm:= 0;
   if l<0 then ll:=l else ll:= 0;
   if nn+mm+ll=0 then else if nn+mm+ll<>n+m+l then p:= nihil;

   { -- first take care of the under world -- }
   for t:=XRoot+1 to nn do
     if odd(trunc(i/TwoPow[n-t])) then p:=PNTR[XKR,p] else p:=PNTR[XKL,p];
   for t:=YRoot+1 to mm do
     if odd(trunc(j/TwoPow[m-t])) then p:=PNTR[YKR,p] else p:=PNTR[YKL,p];
   for t:=ZRoot+1 to ll do
     if odd(trunc(k/TwoPow[l-t])) then p:=PNTR[ZKR,p] else p:=PNTR[ZKL,p];

   { -- and now the upper world -- }
   for t:=1 to n do
     if odd(trunc(i/TwoPow[n-t])) then p:=PNTR[XKR,p] else p:=PNTR[XKL,p];
   for t:=1 to m do
     if odd(trunc(j/TwoPow[m-t])) then p:=PNTR[YKR,p] else p:=PNTR[YKL,p];
   for t:=1 to l do
     if odd(trunc(k/TwoPow[l-t])) then p:=PNTR[ZKR,p] else p:=PNTR[ZKL,p];

   GetPtr:= p;
   { -- ChkPtr(1, n,m,l, i,j,k, p); -- Chk*** }
 end;
end;

procedure GetLocation;
{ -- procedure GetLocation (p :pointer; var n,m,l, i,j,k :integer); -- }
      { -- Find the location for a patch in the grid -- }
begin n:= PNTR[XL,p]; m:= PNTR[YL,p]; l:= PNTR[ZL,p];
      i:= PNTR[XJ,p]; j:= PNTR[YJ,p]; k:= PNTR[ZJ,p];
end;

procedure GetKidType (patch :pointer; var xtyp,ytyp,ztyp :KidType);
begin
   if odd(PNTR[XJ,patch]) then xtyp:=XKR else xtyp:= XKL;
   if odd(PNTR[YJ,patch]) then ytyp:=YKR else ytyp:= YKL;
   if odd(PNTR[ZJ,patch]) then ztyp:=ZKR else ztyp:= ZKL;
end;


procedure SetPlnProperties (p :pointer);
{ -- We use the location of the complete cells
```

```
  -- to determine the location of wall and edges
  -- }
begin
   if (p=nihil) then else
   begin
      { -- completeness is a gift, not a right -- }
      if PPTY[Complete,p] then
      begin
         PPTY[XWall,p]:= true;
         PPTY[YWall,p]:= true;
         PPTY[ZWall,p]:= true;
      end else
      begin
         PPTY[XWall,p]:= PPTY[Complete, PNTR[XNL,p]];
         PPTY[YWall,p]:= PPTY[Complete, PNTR[YNL,p]];
         PPTY[ZWall,p]:= PPTY[Complete, PNTR[ZNL,p]];
      end;

      PPTY[XEdge,p] := false; PPTY[YEdge,p] := false; PPTY[ZEdge,p] := false;

      if PPTY[XWall,p] then
         begin PPTY[YEdge,p] := true; PPTY[ZEdge,p] := true; end;
      if PPTY[YWall,p] then
         begin PPTY[ZEdge,p] := true; PPTY[XEdge,p] := true; end;
      if PPTY[ZWall,p] then
         begin PPTY[XEdge,p] := true; PPTY[YEdge,p] := true; end;

      if not PPTY[XEdge,p] then
             PPTY[XEdge,p] := PPTY[Complete,PNTR[YNL,PNTR[ZNL,p]]];
      if not PPTY[YEdge,p] then
             PPTY[YEdge,p] := PPTY[Complete,PNTR[ZNL,PNTR[XNL,p]]];
      if not PPTY[ZEdge,p] then
             PPTY[ZEdge,p] := PPTY[Complete,PNTR[XNL,PNTR[YNL,p]]];

      { --
      if (PPTY[XEdge,p] or PPTY[YEdge,p] or PPTY[ZEdge,p]) then else
         if PPTY[Complete,PNTR[XNL,PNTR[YNL,PNTR[ZNL,p]]]] then else
            warning(p,'SetPlnPpties',1);
        -- all true patches have the 'Point' property !! -- }
      { -- This patch apparently has no right of existence,
           is it a virtual patch ??? -- }
   end;
end;


procedure SetGrnProperties (p :pointer);
{ -- We use the location of the complete cells
  -- to determine the location of (green) boundaries
  -- }
begin
   if (p=nihil) then else
   begin
      PPTY[XGrnWall,p]:= PPTY[Complete,p] <> PPTY[Complete,PNTR[XNL,p]];
      PPTY[YGrnWall,p]:= PPTY[Complete,p] <> PPTY[Complete,PNTR[YNL,p]];
      PPTY[ZGrnWall,p]:= PPTY[Complete,p] <> PPTY[Complete,PNTR[ZNL,p]];

      PPTY[XGrnEdge,p]:= PPTY[YGrnWall,p] or PPTY[ZGrnWall,p];
      PPTY[YGrnEdge,p]:= PPTY[XGrnWall,p] or PPTY[ZGrnWall,p];
```

```
      PPTY[ZGrnEdge,p]:= PPTY[YGrnWall,p] or PPTY[XGrnWall,p];

      if (not PPTY[XGrnEdge,p]) then PPTY[XGrnEdge,p]
         := PPTY[Complete,p]<>PPTY[Complete,PNTR[ZNL,PNTR[YNL,p]]];
      if (not PPTY[YGrnEdge,p]) then PPTY[YGrnEdge,p]
         := PPTY[Complete,p]<>PPTY[Complete,PNTR[XNL,PNTR[ZNL,p]]];
      if (not PPTY[ZGrnEdge,p]) then PPTY[ZGrnEdge,p]
         := PPTY[Complete,p]<>PPTY[Complete,PNTR[YNL,PNTR[XNL,p]]];

      PPTY[GrnPoint,p]:= PPTY[XGrnEdge,p] or
                         PPTY[YGrnEdge,p] or PPTY[ZGrnEdge,p];
      if not PPTY[GrnPoint,p] then
         PPTY[GrnPoint,p]:= PPTY[Complete,p]<>
         PPTY[Complete,PNTR[XNL,PNTR[YNL,PNTR[ZNL,p]]]];
   end;
end;


procedure SetGrnCell (p :pointer);
begin
   PPTY[GrnCell,p]:= PPTY[Complete,p] and
      ( ( PPTY[XGrnWall,p] <> PPTY[XBdyWall,p] ) or
        ( PPTY[YGrnWall,p] <> PPTY[YBdyWall,p] ) or
        ( PPTY[ZGrnWall,p] <> PPTY[ZBdyWall,p] ) or
        ( PPTY[XGrnWall,PNTR[XNR,p]] <> PPTY[XBdyWall,PNTR[XNR,p]] ) or
        ( PPTY[YGrnWall,PNTR[YNR,p]] <> PPTY[YBdyWall,PNTR[YNR,p]] ) or
        ( PPTY[ZGrnWall,PNTR[ZNR,p]] <> PPTY[ZBdyWall,PNTR[ZNR,p]] ) )
end;


procedure SetBdyProperties (p :pointer);
var
  q, n,m,l, i,j,k :integer;
begin
   SetGrnProperties(p);
   { -- GetLocation(p, n,m,l, i,j,k); -- }
   n:= PNTR[XL,p]; m:= PNTR[YL,p]; l:= PNTR[ZL,p];
   i:= PNTR[XJ,p]; j:= PNTR[YJ,p]; k:= PNTR[ZJ,p];

   if (p=nihil) then else
   if (n+m+l>0) then
   begin { -- We use the boundary structure of the coarser grids -- }
         { -- to determine the boundary structure               -- }
      PPTY[XBdyWall,p]:=
       (PPTY[XBdyWall,PNTR[XF,p]] and (i mod 2 = 0)) or
        PPTY[XBdyWall,PNTR[YF,p]] or PPTY[XBdyWall,PNTR[ZF,p]];
      PPTY[YBdyWall,p]:=
       (PPTY[YBdyWall,PNTR[YF,p]] and (j mod 2 = 0)) or
        PPTY[YBdyWall,PNTR[XF,p]] or PPTY[YBdyWall,PNTR[ZF,p]];
      PPTY[ZBdyWall,p]:=
       (PPTY[ZBdyWall,PNTR[ZF,p]] and (k mod 2 = 0)) or
        PPTY[ZBdyWall,PNTR[YF,p]] or PPTY[ZBdyWall,PNTR[XF,p]];
      { -- this can be implemented more efficiently -- }

      PPTY[XBdyEdge,p]:= PPTY[YBdyWall,p] or PPTY[ZBdyWall,p];
      PPTY[YBdyEdge,p]:= PPTY[XBdyWall,p] or PPTY[ZBdyWall,p];
      PPTY[ZBdyEdge,p]:= PPTY[YBdyWall,p] or PPTY[XBdyWall,p];
      if not  PPTY[XBdyEdge,p] then PPTY[XBdyEdge,p]:=
```

```
         PPTY[XBdyEdge,PNTR[XF,p]] or
         (PPTY[XBdyEdge,PNTR[YF,p]] and (j mod 2 = 0)) or
         (PPTY[XBdyEdge,PNTR[ZF,p]] and (k mod 2 = 0));
      if not  PPTY[YBdyEdge,p] then PPTY[YBdyEdge,p]:=
         PPTY[YBdyEdge,PNTR[YF,p]] or
         (PPTY[YBdyEdge,PNTR[XF,p]] and (i mod 2 = 0)) or
         (PPTY[YBdyEdge,PNTR[ZF,p]] and (k mod 2 = 0));
      if not  PPTY[ZBdyEdge,p] then PPTY[ZBdyEdge,p]:=
         PPTY[ZBdyEdge,PNTR[ZF,p]] or
         (PPTY[ZBdyEdge,PNTR[YF,p]] and (j mod 2 = 0)) or
         (PPTY[ZBdyEdge,PNTR[XF,p]] and (i mod 2 = 0));

      PPTY[BdyPoint,p]:= PPTY[XBdyEdge,p] or
                         PPTY[YBdyEdge,p] or PPTY[ZBdyEdge,p];
      if not PPTY[BdyPoint,p] then PPTY[BdyPoint,p]:=
         (PPTY[BdyPoint,PNTR[XF,p]] and (i mod 2 = 0)) or
         (PPTY[BdyPoint,PNTR[YF,p]] and (j mod 2 = 0)) or
         (PPTY[BdyPoint,PNTR[ZF,p]] and (k mod 2 = 0));
   end
   else if (n+m+l=0) then
      for q:= FstBdyPpt to LstBdyPpt do PPTY[q,p]:= PPTY[q+GrnShift,p]
   else
      error(p,'SetBdyPpties',99);

   SetGrnCell(p); for q:= XNL to ZNL do SetGrnCell(PNTR[q,p]);
end;


function MakePatch (n,m,l, i,j,k : integer; p :pointer): pointer;
{ -- The patch is made only if three parent patches are available ! -- }
{ -- Takes care for ALL the pointers      -- }
{ -- Does NOT take care of any properties -- }
const n0=0; m0=0; l0=0;
var
   q :integer;
   xtyp,ytyp,ztyp :KidType;
   dadX,dadY,dadZ, kid,
   nxr, nxl, nyr, nyl, nzr, nzl :pointer;
begin
   { -- ChkPtr(2, n,m,l, i,j,k, p); Chk*** -- }
   if (p=nihil) then
   begin
      { -- We take care of possible cyclic numbering  -- }
      { -- GetLocation(p, n,m,l, i,j,k); -- DOESN'T WORK HERE -- }
      if XCycl then begin if n<0 then else i:= i mod (XSize*TwoPow[n]) end;
      if YCycl then begin if m<0 then else j:= j mod (YSize*TwoPow[m]) end;
      if ZCycl then begin if l<0 then else k:= k mod (ZSize*TwoPow[l]) end;

      { -- determine the fathers -- }
      kid:= nihil;
      dadX:=GetPtr(n-1,m,l, trunc(i/2),j,k);
      dadY:=GetPtr(n,m-1,l, i,trunc(j/2),k);
      dadZ:=GetPtr(n,m,l-1, i,j,trunc(k/2));
      if ((dadX=nihil) and (n>n0)) then else
      if ((dadY=nihil) and (m>m0)) then else
      if ((dadZ=nihil) and (l>l0)) then else
{ -- recursive creation of parents is not feasible for various reasons ! -- }
      begin
```

```
LastSpace:= LastSpace-1;
repeat LastSpace:= LastSpace+1
    until PPTY[Dead,LastSpace] or (LastSpace>NumberOfPatches);
{ -- LastSpace now points to the first empty space for a new patch -- }
if LastSpace>NumberOfPatches then NumberOfPatches:= LastSpace;

if NumberOfPatches > MNOP then error(0,'MakePatch',1)
                          else kid:= LastSpace;
{ -- NO kids, NO properties -- }
for q:= FstPtr to LstPtr do PNTR[q,kid]:= nihil;
for q:= FstPpt to LstPpt do PPTY[q,kid]:= false;
if n+m+l>0 then GeometryOK:= true;

PNTR [XL,kid]:=    n; PNTR [YL,kid]:=    m; PNTR [ZL,kid]:=    l;
PNTR [XJ,kid]:=    i; PNTR [YJ,kid]:=    j; PNTR [ZJ,kid]:=    k;
PNTR [XF,kid]:= dadX; PNTR [YF,kid]:= dadY; PNTR [ZF,kid]:= dadZ;

GetKidType(kid,xtyp,ytyp,ztyp);
{ -- dat kan efficienter -- }
if (dadX=nihil) then else PNTR[xtyp,dadX]:= kid;
if (dadY=nihil) then else PNTR[ytyp,dadY]:= kid;
if (dadZ=nihil) then else PNTR[ztyp,dadZ]:= kid;


{ -- Now  we take care of neighbours -- }
if n=n0 then
     begin nxr:= GetPtr(n,m,l, i+1,j,k);
           nxl:= GetPtr(n,m,l, i-1,j,k);
     end else { -- father exists -- }
case xtyp of
 XKL:begin nxl:= PNTR[XKR,PNTR[XNL,dadX]];
           nxr:= PNTR[XKR,         dadX ];
     end;
 XKR:begin nxr:= PNTR[XKL,PNTR[XNR,dadX]];
           nxl:= PNTR[XKL,         dadX ];
     end;
end;
if m=m0 then
     begin nyr:= GetPtr(n,m,l, i,j+1,k);
           nyl:= GetPtr(n,m,l, i,j-1,k);
     end else { -- father exists -- }
case ytyp of
 YKL:begin nyl:= PNTR[YKR,PNTR[YNL,dadY]];
           nyr:= PNTR[YKR,         dadY ];
     end;
 YKR:begin nyr:= PNTR[YKL,PNTR[YNR,dadY]];
           nyl:= PNTR[YKL,         dadY ];
     end;
end;
if l=l0 then
     begin nzr:= GetPtr(n,m,l, i,j,k+1);
           nzl:= GetPtr(n,m,l, i,j,k-1);
     end else { -- father exists -- }
case ztyp of
 ZKL:begin nzl:= PNTR[ZKR,PNTR[ZNL,dadZ]];
           nzr:= PNTR[ZKR,         dadZ ];
     end;
 ZKR:begin nzr:= PNTR[ZKL,PNTR[ZNR,dadZ]];
```

```
                  nzl:= PNTR[ZKL,         dadZ ];
              end;
          end;

          if (PNTR[XNL,nxr]+PNTR[XNR,nxl]+PNTR[YNL,nyr]+
              PNTR[YNR,nyl]+PNTR[ZNL,nzr]+PNTR[ZNR,nzl]<>0)
             then error(kid,'MakePatch',12);
              { -- warning if the neighbours know each other already -- }

          PNTR[XNR,kid]:= nxr; if nxr<>nihil then PNTR[XNL,nxr]:= kid;
          PNTR[XNL,kid]:= nxl; if nxl<>nihil then PNTR[XNR,nxl]:= kid;
          PNTR[YNR,kid]:= nyr; if nyr<>nihil then PNTR[YNL,nyr]:= kid;
          PNTR[YNL,kid]:= nyl; if nyl<>nihil then PNTR[YNR,nyl]:= kid;
          PNTR[ZNR,kid]:= nzr; if nzr<>nihil then PNTR[ZNL,nzr]:= kid;
          PNTR[ZNL,kid]:= nzl; if nzl<>nihil then PNTR[ZNR,nzl]:= kid;

      end;
      MakePatch:= kid;
      { -- ChkPtr(44, n,m,l, i,j,k, kid);  Chk*** -- }
   end else
      MakePatch:= p;
end;


procedure MakeCell (n,m,l, i,j,k :integer; patch :pointer);
{ -- A cell is a complete patch with sufficient neighbours.        -- }
{ -- A cell can be made complete only if it has 3 complete fathers -- }
var
   q :integer;
   p :array[0..7]of pointer;
begin
   { -- if patch<>nihil then ChkPtr(7, n,m,l, i,j,k, patch); Chk*** -- }
   if patch=nihil then p[0]:= GetPtr(n,m,l, i,j,k) else p[0]:= patch;
   { --  that is the kid or nihil-- }
   if PPTY[Complete,p[0]] then { -- the cell already exists -- } else
   begin
      p[0]:= MakePatch (n,m,l, i,j,k ,p[0]);
      PPTY[Complete,p[0]]:= ((n=0) or PPTY[Complete,PNTR[XF,p[0]]]) and
                            ((m=0) or PPTY[Complete,PNTR[YF,p[0]]]) and
                            ((l=0) or PPTY[Complete,PNTR[ZF,p[0]]]);
      if PPTY[Complete,p[0]] then
         begin
            p[1]:= MakePatch (n,m,l, i+1,j  ,k  ,PNTR[XNR,p[0]]);
            p[2]:= MakePatch (n,m,l, i  ,j+1,k  ,PNTR[YNR,p[0]]);
            p[3]:= MakePatch (n,m,l, i  ,j  ,k+1,PNTR[ZNR,p[0]]);
            p[4]:= MakePatch (n,m,l, i  ,j+1,k+1,PNTR[YNR,p[3]]);
            p[5]:= MakePatch (n,m,l, i+1,j  ,k+1,PNTR[ZNR,p[1]]);
            p[6]:= MakePatch (n,m,l, i+1,j+1,k  ,PNTR[XNR,p[2]]);
            p[7]:= MakePatch (n,m,l, i+1,j+1,k+1,PNTR[XNR,p[4]]);
            if p[0]*p[1]*p[2]*p[3]*p[4]*p[5]*p[6]*p[7] = nihil
               then error(p[0],'MakeCell',3);

            for q:=0 to 7 do SetPlnProperties(p[q]);
            for q:=0 to 7 do SetBdyProperties(p[q]);
         end
      else
         begin { -- this happens for GrnPoints ! --}
            SetPlnProperties(p[0]);
```

```
            SetBdyProperties(p[0]);
         end;
   end;
end;


procedure MakeKid (ktyp: KidType; daddy: pointer);
{ -- A cell is a complete patch with sufficient neighbours.        -- }
{ -- A cell can be made complete only if it has 3 complete fathers -- }
var
   i,j,k,n,m,l :integer;
begin
   if (ktyp < FstKid) or (ktyp>LstKid) then error(daddy,'MakeKid',1);
   GetLocation(daddy, n,m,l, i,j,k);
   if (n<0) or (m<0) or (l<0) then error(daddy,'MakeKid',2);
   case ktyp of
      XKL: begin n:=n+1; i:= 2*i  ; end;
      XKR: begin n:=n+1; i:= 2*i+1; end;
      YKL: begin m:=m+1; j:= 2*j  ; end;
      YKR: begin m:=m+1; j:= 2*j+1; end;
      ZKL: begin l:=l+1; k:= 2*k  ; end;
      ZKR: begin l:=l+1; k:= 2*k+1; end;
   end;
   MakeCell (n,m,l, i,j,k, PNTR[ktyp, daddy]);
end;


procedure MakeOffspring  (patch: pointer);
{ -- Takes only care of all kids -- }
begin
  if PPTY[Complete,patch] then
    begin
      if PPTY[PregnantX,patch] then
         begin MakeKid(XKL, patch); MakeKid(XKR, patch);
               PPTY[PregnantX,patch]:=false;
         end;
      if PPTY[PregnantY,patch] then
         begin MakeKid(YKL, patch); MakeKid(YKR, patch);
               PPTY[PregnantY,patch]:=false;
         end;
      if PPTY[PregnantZ,patch] then
         begin MakeKid(ZKL, patch); MakeKid(ZKR, patch);
               PPTY[PregnantZ,patch]:=false;
         end;
    end;
end;


procedure RemovePatch(patch :pointer);
{ -- If possible, this routine removes a patch from the system.    -- }
{ -- It is possible if its point is not part of a (complete) cell -- }
{ -- and if it is not responsible for kid patches -- }
var
   i :integer;
   xtyp, ytyp, ztyp :KidType;
begin
   if        patch = nihil then else
   if PPTY[Complete,patch] then else
```

```
  if PPTY[GrnPoint,patch] then else
  if ((PNTR[XKL,patch]=nihil) and (PNTR[XKR,patch]=nihil) and
      (PNTR[YKL,patch]=nihil) and (PNTR[YKR,patch]=nihil) and
      (PNTR[ZKL,patch]=nihil) and (PNTR[ZKR,patch]=nihil)) then
  begin
     { -- The relation with Neighbours is closed -- }
     PNTR[XNL,PNTR[XNR,patch]]:= nihil;
     PNTR[XNR,PNTR[XNL,patch]]:= nihil;
     PNTR[YNL,PNTR[YNR,patch]]:= nihil;
     PNTR[YNR,PNTR[YNL,patch]]:= nihil;
     PNTR[ZNL,PNTR[ZNR,patch]]:= nihil;
     PNTR[ZNR,PNTR[ZNL,patch]]:= nihil;

     { -- The relation with Parent is closed -- }
     GetKidType(patch, xtyp,ytyp,ztyp);
     PNTR[xtyp, PNTR[XF,patch]]:= nihil;
     PNTR[ytyp, PNTR[YF,patch]]:= nihil;
     PNTR[ztyp, PNTR[ZF,patch]]:= nihil;
     if patch<LastSpace then LastSpace:=patch;

     { -- All indices, pointers and properties are removed -- }
     for i:= FstIdx to LstIdx do PNTR[i,patch]:= nowhere;
     for i:= FstPtr to LstPtr do PNTR[i,patch]:= nihil;
     for i:= FstPpt to LstPpt do PPTY[i,patch]:= false;

     PPTY[Dead,patch]:= true;
  end;
end;


procedure RemoveCell(patch :pointer);
{ -- If possible (if there are no kid cells )
  -- this routine removes a cell from the system.
  -- I.e. no longer a complete cell exists
  -- possibly it remains as an incomplete patch -- }
var
  skip :boolean;
  n :integer;
  p :array [0..7]of pointer;
begin
   skip:= not PPTY[Complete,patch];
   for n:= FstKid to LstKid do skip:= skip or PPTY[Complete,PNTR[n,patch]];

   if skip then else
   if GeometryOK and (PNTR[XL,patch]+PNTR[YL,patch]+PNTR[ZL,patch]=0) then
      { -- if the geometry has been established,
        -- no change has to be made on level 0    -- }
      warning(patch,'RemoveCell',0)
   else
   begin
      PPTY[Complete,patch]:= false;
      p[0]:= patch;
      p[1]:= PNTR[XNR,patch];
      p[2]:= PNTR[YNR,patch];
      p[3]:= PNTR[ZNR,patch];
      p[4]:= PNTR[YNR,PNTR[ZNR,patch]];
      p[5]:= PNTR[ZNR,PNTR[XNR,patch]];
      p[6]:= PNTR[XNR,PNTR[YNR,patch]];
```

```
      p[7]:= PNTR[XNR,PNTR[YNR,PNTR[ZNR,patch]]];
      if p[0]*p[1]*p[2]*p[3]*p[4]*p[5]*p[6]*p[7]=nihil
         then error(patch,'RemoveCell',1);

      for n:= 0 to 7 do SetPlnProperties(p[n]);
      for n:= 0 to 7 do SetBdyProperties(p[n]);
      for n:= 0 to 7 do RemovePatch(p[n]);
   end
end;


procedure RemoveOffspring(patch :pointer);
{ -- If possible this routine removes the 6 kids of daddy and it
  -- adapts the data structure correspondingly. It is only possible
  -- if the kids are sentenced. -- }
var
   kt :KidType;
begin
   for kt:= FstKid to LstKid do
   if PPTY[Sentenced,PNTR[kt,patch]] then RemoveCell(PNTR[kt,patch]);
end;


procedure ScanGrid (n,m,l :integer; myorder :order;
                    procedure DoIt (p:pointer));
var
    RootLevel, ScanThisLevel, lev, nn, mm, ll,
    id, chk, i,j :integer;
    ii, IPTR: array[LNOL..MNOL]of integer;
    KeepScanOrder : array[LNOL..MNOL,1..2]of KidType;
begin
     chk:=0; { -- check if myorder is legal -- } for i:= 1 to 3 do
     case myorder[i] of
       XKL, XKR: chk:= chk+1; YKL, YKR: chk:= chk+2; ZKL, ZKR: chk:= chk+4;
     end; if chk <> 7 then error(0,'ScanGrid',1);

     ScanThisLevel:= n+m+l;
{ -- This first part constructs an array 'KeepScanOrder'
  -- that determines the way in which the grid (n,m,l) is scanned -- }
     id:=XRoot+YRoot+ZRoot;
     if n<0 then nn:=n else nn:= 0;
     if m<0 then mm:=m else mm:= 0;
     if l<0 then ll:=l else ll:= 0;
     if nn+mm+ll=0 then else if nn+mm+ll<>n+m+l then error(nihil,'ScanGrid',0);

     for i:= 1 to 3 do
     case myorder[i] of
     XKL, XKR: for j:= XRoot+1 to nn do
               begin id:= id+1;
                     KeepScanOrder[id,1]:= myorder[i];
                     KeepScanOrder[id,2]:= XKR+XKL-myorder[i];
               end;
     YKL, YKR: for j:= YRoot+1 to mm do
               begin id:= id+1;
                     KeepScanOrder[id,1]:= myorder[i];
                     KeepScanOrder[id,2]:= YKR+YKL-myorder[i];
               end;
     ZKL, ZKR: for j:= ZRoot+1 to ll do
```

```
   ScanGrid(0,0,0,NormalOrder,SetBdyProperties);
end;


procedure MakeFamily (n,m,l, i,j,k :integer);
{ -- this procedure creates a cell, together with  -- }
{ -- all the necessary parents                     -- }
 var lev :integer;
   procedure MakeIt (nn,mm,ll :integer);
   begin  MakeCell (nn,mm,ll,
          trunc(i/TwoPow[n-nn]),trunc(j/TwoPow[m-mm]),trunc(k/TwoPow[l-ll]),
          nihil);
   end;
begin for lev:=0 to n+m+l do  ScanLevel(lev,0,n,0,m,0,l,MakeIt); end;



procedure ShowPtr (patch: pointer);
var
   k :integer;
begin
   write(patch:4,' @');
   for k:= FstIdx to LstIdx do if PNTR[k,patch]=nowhere
       then write(' ***') else write(PNTR[k,patch]:4); write(',');
   for k:=     XF to     ZF do write(PNTR[k,patch]:4); write(',');
   for k:= FstKid to LstKid do write(PNTR[k,patch]:4); write(',');
  writeln;
   for k:= 1 to 44 do write(' ');
   for k:= FstNgb to LstNgb do write(PNTR[k,patch]:4); write(',');
  writeln;
end;

procedure ShowPpt (patch: pointer);
var
   k :integer;
   str:array[1..2] of char;
begin
   write(patch:4,' @');
   for k:= FstPpt to LstPpt do
     begin if PPTY[k,patch] then
       case k of
         1: str:= 'C_';
 2: str:= 'WX';  3: str:= 'WY';  4: str:= 'WZ';
         5: str:= 'EX';  6: str:= 'EY';  7: str:= 'EZ';
         8: str:= 'BX';  9: str:= 'BY'; 10: str:= 'BZ';
        11: str:= 'bx'; 12: str:= 'by'; 13: str:= 'bz';
        14: str:= 'BP'; 15: str:= 'BC';
        16: str:= 'GX'; 17: str:= 'GY'; 18: str:= 'GZ';
        19: str:= 'gx'; 20: str:= 'gy'; 21: str:= 'gz';
        22: str:= 'GP'; 23: str:= 'GC';
        24: str:= 'Px'; 25: str:= 'Py'; 26: str:= 'Pz';
        27: str:= 'Sd'; 28: str:= 'Dd';
      end else str:= '__';
      write(str);
    end;
  writeln;
end;
```

```
procedure ShowPatch;
{ -- procedure ShowPatch (patch: pointer); -- }
begin if (patch=nihil)
         then writeln('ShowPatch(NIHIL)')
         else begin ShowPpt(patch); ShowPtr(patch); end;
end;

procedure ShowGrid (n,m,l :integer);
begin ScanGrid(n,m,l,NormalOrder,ShowPatch) end;

procedure ShowLevel(lev :integer);
begin ScanLevel(lev,XRoot,MNOL,YRoot,MNOL,ZRoot,MNOL,ShowGrid) end;

procedure ShowAll;
var p :pointer;
begin for p:= 1 to NumberOfPatches do ShowPatch(p); end;


procedure DumpAll ( name : strng);
var
   k,p :integer;
   dumpfile :text;
begin
   rewrite(dumpfile,name);
   for p:= 1 to NumberOfPatches do
   if PPTY[Dead,p] then else
   begin
      write(dumpfile, PNTR[XL,p]:4,PNTR[YL,p]:4,PNTR[ZL,p]:4,
                      PNTR[XJ,p]:6,PNTR[YJ,p]:6,PNTR[ZJ,p]:6);
      for k:= FstPpt to LstPpt do
      if PPTY[k,p] then write(dumpfile,' T')
                   else write(dumpfile,' F');
      for k:= 1 to MNOD do write (dumpfile,DATA[k,p]:8:4);
      writeln(dumpfile);
   end;
end;
```

A.2 About the FORTRAN implementation

In the Fortran implementation all data about the data structure are collected in a labeled COMMON BLOCK labeled `/DatGlb/`. Together with other global variables (and constants) this is housed within the include file 'basis3.i'. This include file is to be included in the main program and each routine that makes use of the data structure. The arrays, declared in the include file, containing the bulk of the data structure, are: the integer array PNTR, dimensioned PNTR (FstIdx:LstPtr, 0:MNOP), the logical array PPTY (FstPpt:LstPpt, 0:MNOP); and the double precision array MYDATA (1:MNOD, 0:MNOP). These arrays contain the dynamic part of the data structure. The parameters FstPtr, LstPtr, FstPpt, LstPpt, MNOD, and MNOP can be adapted by the user for his own purposes. Other global variables, collected in the common block /DatGlb/, are a.o. RtLv, LstSpa, NOP, XSize, YSize, ZSize[1]. The default ranking order in which the data structure is scanned, is given by the order array NrmOrd (Normal Order).

[1]RtLv = RootLevel; LstSpa = Last Space; NOP = number of points; (XSize, YSize, ZSize) denotes the number of cells on the zero level.

The routines that are to be used to handle the data structure are collected in the file 'basis3.f'. The text of the include file is given in Section A.3. The meaning of the constants and variables is explained in Section A.4, where also a description of important subroutines is given in the form of Fortran comment lines.

A.3 The FORTRAN include file

```
c
c     begin of include file for the data structure BASIS3
      integer        MNOP,            MNOD,
     +               MNOL,            LNOL
      parameter      (MNOP   = 30000, MNOD   =     15,
     +               MNOL    =    30, LNOL   =    -12)
c
      integer        nihil,           void
      parameter      (nihil  =     0, void   =   -999)
c
      integer        XL,              YL,              ZL,
     +               XJ,              YJ,              ZJ
      parameter      (XL     =     1, YL     =      2, ZL     =      3,
     +               XJ      =     4, YJ     =      5, ZJ     =      6)
c
      integer        FstIdx,          LstIdx
      parameter      (FstIdx =    XL, LstIdx =     ZJ)
c
      integer        XF,              YF,              ZF,
     +               XKL,             YKL,             ZKL,
     +               XKR,             YKR,             ZKR
      parameter      (XF     =     7, YF     =      8, ZF     =      9,
     +               XKL     =    10, YKL    =     11, ZKL    =     12,
     +               XKR     =    13, YKR    =     14, ZKR    =     15)
c
      integer        FstKid,          LstKid
      parameter      (FstKid =   XKL, LstKid =    ZKR)
c
      integer        XNL,             YNL,             ZNL,
     +               XNR,             YNR,             ZNR
      parameter      (XNL    =    16, YNL    =     17, ZNL    =     18,
     +               XNR     =    21, YNR    =     20, ZNR    =     19)
c
      integer        FstNgb,          LstNgb,
     +               FstPtr,          LstPtr
      parameter      (FstNgb =   XNL, LstNgb =    XNR,
     +               FstPtr  =    XF, LstPtr =    XNR)
c
      integer        Compl,
     +               XWall,           YWall,           ZWall,
     +               XEdge,           YEdge,           ZEdge
      parameter      (Compl  =     1,
     +               XWall   =     2, YWall  =      3, ZWall  =      4,
     +               XEdge   =     5, YEdge  =      6, ZEdge  =      7)
c
      integer        XBdyWa,          YBdyWa,          ZBdyWa,
     +               XBdyEd,          YBdyEd,          ZBdyEd,
     +               BdyPnt
      parameter      (XBdyWa =     8, YBdyWa =      9, ZBdyWa =     10,
     +               XBdyEd  =    11, YBdyEd =     12, ZBdyEd =     13,
```

```
     +                BdyPnt =    14)
c
c     integer         BdyCel
c     parameter      (BdyCel =    15)
c
      integer         XGrnWa,         YGrnWa,         ZGrnWa,
     +                XGrnEd,         YGrnEd,         ZGrnEd,
     +                GrnPnt,         GrnCel
      parameter      (XGrnWa =    16, YGrnWa =    17, ZGrnWa =    18,
     +                XGrnEd =    19, YGrnEd =    20, ZGrnEd =    21,
     +                GrnPnt =    22, GrnCel =    23)
c
      integer         PrgntX,         PrgntY,         PrgntZ,
     +                Sntncd,         Dead
      parameter      (PrgntX =    24, PrgntY =    25, PrgntZ =    26,
     +                Sntncd =    27, Dead   =    28)
c
      integer         FstBPp,         LstBPp,
     +                GrnShi,
     +                FstPpt,         LstPpt
      parameter      (FstBPp = XBdyWa,LstBPp = BdyPnt,
     +                GrnShi = GrnPnt - BdyPnt,
     +                FstPpt = Compl, LstPpt    = Dead)
c
      double precision MYDATA(1:MNOD, 0:MNOP)
      integer         XSize,          YSize,          ZSize,
     +                XRoot,          YRoot,          ZRoot,
     +                RtPtr,
     +                LstSpa,         NOP
      integer         NrmOrd(1:3)
      integer         TwoPow(0:MNOL)
      integer         PNTR(FstIdx:LstPtr, 0:MNOP)
      logical         GeomOK,
     +                XCycl,          YCycl,          ZCycl
      logical         PPTY(FstPpt:LstPpt, 0:MNOP)
      common /DatGlb/
     +                MYDATA,
     +                XSize,          YSize,          ZSize,
     +                XRoot,          YRoot,          ZRoot,
     +                RtPtr,
     +                LstSpa,         NOP,
     +                NrmOrd,
     +                TwoPow,
     +                PNTR,
     +                GeomOK,
     +                XCycl,          YCycl,          ZCycl,
     +                PPTY
c     end of include file for the data structure BASIS3
c
```

A.4 The FORTRAN implementation manual

```
c
c  DESCRIPTION OF INCLUDE FILE  + + + + + + + + + + + + + + + + + + + + +
c
c  In order to condense the code and to enhance its clarity, a so
c  called include-statement has been used:
c     include 'basis3.i'
```

```
c In this way, variables and constants introduced for the data
c structure, have the same symbolic name throughout the code.
c The include statement is nonstandard syntax and dependent on the
c FORTRAN-compiler in use. However, a version of the code, remaining
c within the constraints of the Fortran 77 standard, can easily be
c obtained by substituting the include file 'basis3.i' for the said
c include-statement in the subroutines.
c The include file includes the Labeled COMMON Block /DatGlb/.
c Firstly, we describe the meaning of important global parameters.
c Secondly, we describe the meaning of important global variables
c as they are found in /DatGlb/.
c
c
c DIMENSIONS TO BE ADJUSTED BY THE USER TO THE SIZE OF THE SYSTEM USED
c MNOP      Maximum Number Of Patches
c MNOD      Maximum Number Of Data at a patch
c The values MNOP, MNOD are integer parameter-values and should be set
c by the user, according to his needs.
c MNOL      Maximum Number Of grid-Levels
c LNOL      Lowest Number Of grid-Levels
c The values MNOL, LNOL are integer parameter-values and can, if needed,
c be adapted by the user.
c
c POINTER VALUES
c nihil     the nil pointer
c void      nowhere in the structure
c
c INTEGER ARRAY INDICES
c - these render geometric information about a patch -
c (see Table 8 in the report)
c XL        Level index of refinement in the X-direction
c YL        Level index of refinement in the Y-direction
c ZL        Level index of refinement in the Z-direction
c
c XJ        Coordinate index - integer representation of X-coordinate
c YJ        Coordinate index - integer representation of Y-coordinate
c ZJ        Coordinate index - integer representation of Z-coordinate
c
c FstIdx    First Index
c           Lowest Index of the set (XL, YL, ZL, XJ, YJ, ZJ)
c LstIdx    Last Index
c           Highest Index of the set (XL, YL, ZL, XJ, YJ, ZJ)
c
c more INTEGER ARRAY INDICES
c - these indicate pointers to related patches -
c (see the Tables 2, 3 and 4 in the report)
c XF        pointer to X-Father
c YF        pointer to Y-Father
c ZF        pointer to Z-Father
c
c XKL       pointer to Lefthand X-Kid (kid with even X-coordinate)
c YKL       pointer to Lefthand Y-Kid (kid with even Y-coordinate)
c ZKL       pointer to Lefthand Z-Kid (kid with even Z-coordinate)
c
c XKR       pointer to Righthand X-Kid (kid with odd X-coordinate)
c YKR       pointer to Righthand Y-Kid (kid with odd Y-coordinate)
c ZKR       pointer to Righthand Z-Kid (kid with odd Z-coordinate)
c
```

```
c  FstKid   First Kid
c           Lowest index of the set of indices for the Kids, i.e.
c           Minimum (XKL, YKL, ZKL, XKR, YKR, ZKR)
c  LstKid   Last Kid
c           Highest index of the set of indices for the Kids, i.e.
c           Maximum (XKL, YKL, ZKL, XKR, YKR, ZKR)
c
c  XNL      pointer to Lefthand X-Neighbour (negative direction)
c  YNL      pointer to Lefthand Y-Neighbour (negative direction)
c  ZNL      pointer to Lefthand Z-Neighbour (negative direction)
c
c  XNR      pointer to Righthand X-Neighbour (positive direction)
c  YNR      pointer to Righthand Y-Neighbour (positive direction)
c  ZNR      pointer to Righthand Z-Neighbour (positive direction)
c
c  FstNgb   First Neighbour
c           Lowest index of the set of indices for pointers to the
c           Neighbours, i.e. Minimum (XNL, YNL, ZNL, XNR, YNR, ZNR)
c  LstNgb   Last Neighbour
c           Highest index of the set of indices for pointers to the
c           Neighbours, i.e. Maximum (XNL, YNL, ZNL, XNR, YNR, ZNR)
c
c  FstPtr   First Pointer
c           Lowest index of the set of indices for pointers to Fathers,
c           Kids and Neighbours, i.e.
c           Minimum (XF, YF, ZF, FstKid, FstNgb)
c  LstPtr   Last Pointer
c           Highest index of the set of indices for pointers to Fathers,
c           Kids and Neighbours, i.e.
c           Maximum (XF, YF, ZF, LstKid, LstNgb)
c
c  LOGICAL ARRAY INDICES
c  - These INDICES render whether the corresponding properties are
c    satisfied (TRUE), or not (FALSE) -
c  (see the Tables 6 and 7 in the report)
c  Compl    Complete
c           the patch represents a volume with contents
c  XWall    a Wall perpendicular to X exists
c  YWall    a Wall perpendicular to Y exists
c  ZWall    a Wall perpendicular to Z exists
c
c  XEdge    an Edge along X exists
c  YEdge    an Edge along Y exists
c  ZEdge    an Edge along Z exists
c
c  XBdyWa   X Boundary Wall
c           a Wall exists perpendicular to X and part of the Boundary
c           of the domain
c  YBdyWa   Y Boundary Wall
c           a Wall exists perpendicular to Y and part of the Boundary
c           of the domain
c  ZBdyWa   Z Boundary Wall
c           a Wall exists perpendicular to Z and part of the Boundary
c           of the domain
c
c  XBdyEd   X Boundary Edge
c           an Edge exists along X and part of the Boundary of the domain
c  YBdyEd   Y Boundary Edge
```

```
c             an Edge exists along Y and part of the Boundary of the domain
c  ZBdyEd     Z Boundary Edge
c             an Edge exists along Z and part of the Boundary of the domain
c  BdyPnt     Boundary Point
c             a Point exists part of the Boundary of the domain
c
c  XGrnWa     X Green Wall
c             the wall perpendicular to X is Green: i.e.
c             either this patch or its XNL-neighbour is not Complete
c             (e.g. at the boundary of the domain or of a local
c             refinement)
c  YGrnWa     Y Green Wall
c             the wall perpendicular to Y is Green: i.e.
c             either this patch or its YNL-neighbour is not Complete
c             (e.g. at the boundary of the domain or of a local
c             refinement)
c  ZGrnWa     Z Green Wall
c             the wall perpendicular to Z is Green: i.e.
c             either this patch or its ZNL-neighbour is not Complete
c             (e.g. at the boundary of the domain or of a local
c             refinement)
c
c  XGrnEd     X Green Edge
c             an Edge along X is part of a Green boundary
c  YGrnEd     Y Green Edge
c             an Edge along Y is part of a Green boundary
c  ZGrnEd     Z Green Edge
c             an Edge along Z is part of a Green boundary
c
c  GrnPnt     Green Point
c             Point at the patch is part of a Green boundary
c
c  GrnCel     Green Cell
c             a Complete Cell that is not at the domain boundary and
c             of which at least one of its six walls is Green
c
c             FLAGS
c  PrgntX     Pregnant in X
c             this flag indicates that the cell should be refined in
c             the X-direction at the first opportunity
c  PrgntY     Pregnant in Y
c             this flag indicates that the cell should be refined in
c             the Y-direction at the first opportunity
c  PrgntZ     Pregnant in Z
c             this flag indicates that the cell should be refined in
c             the Z-direction at the first opportunity
c
c  Sntncd     Sentenced
c             this flag indicates that the cell should be removed
c             at the first opportunity
c
c  Dead       Dead
c             the patch has passed away and its number is free for
c             re-use
c
c             BOUNDS AND SHIFT
c  FstBPp     First Boundary Property
c             Lowest index of the set of indices for the Boundary
```

```
c           Properties
c  LstBPp   Last Boundary Property
c           Highest index of the set of indices for the Boundary
c           Properties
c
c  GrnShi   Green Shift
c           Shift (integer number) between index to Boundary Wall/
c           Edge/Point property and the corresponding Green property,
c           e.g. GrnPnt = BdyPnt + GrnShi
c
c  FstPpt   First Property
c           Lowest index of the set of indices for all Properties
c  LstPpt   Last Property
c           Highest index of the set of indices for all Properties
c
c
c  LABELED COMMON BLOCK FOR THE DATA STRUCTURE
c
c  Global properties and the bulk of the data structure are passed on
c  to different parts of the program by the Labeled COMMON Block /DatGlb/.
c
c     double precision MYDATA(1:MNOD,      0:MNOP)
c     integer          XSize,          YSize,          ZSize,
c    +                 XRoot,          YRoot,          ZRoot,
c    +                 RtPtr,
c    +                 LstSpa,         NOP
c     integer          NrmOrd(1:3)
c     integer          TwoPow(0:MNOL)
c     integer          PNTR(FstIdx:LstPtr, 0:MNOP)
c     logical          GeomOK,
c    +                 XCycl,          YCycl,          ZCycl
c     logical          PPTY(FstPpt:LstPpt, 0:MNOP)
c     common /DatGlb/
c    +                 MYDATA,
c    +                 XSize,          YSize,          ZSize,
c    +                 XRoot,          YRoot,          ZRoot,
c    +                 RtPtr,
c    +                 LstSpa,         NOP,
c    +                 NrmOrd,
c    +                 TwoPow,
c    +                 PNTR,
c    +                 GeomOK,
c    +                 XCycl,          YCycl,          ZCycl,
c    +                 PPTY
c
c  We now describe the meaning of the global variables as they are
c  found in /DatGlb/.
c
c  BULK
c  The bulk of the data structure is kept in the arrays:
c
c     double precision MYDATA(1:MNOD,      0:MNOP)
c
c           These are the numerical data (e.g. velocity, temperature,
c           etc.) residing in the patches.
c
c           The values MNOP, MNOD are integer parameter-values and
c           should be set by the user, according to his needs (see
```

```
c           Section 4.2 of the report).
c
c    integer          PNTR(FstIdx:LstPtr, 0:MNOP)
c
c           These are the pointers arising from the patches, pointing
c           to other patches like neighbours, kids and fathers.
c           Their actual values will be of no concern to the user of
c           the code.
c
c    logical          PPTY(FstPpt:LstPpt, 0:MNOP)
c
c           These are the properties of the patches. They hold
c           information on the discrete geometry (and possibly its
c           next update) of the problem.
c
c           The user can summon this values as a means to interrogate
c           the data structure about its status. The user is not
c           supposed to set or write these data, except for the
c           entries identified by the flags: PrgntX, PrgntY, PrgntZ
c           and Sntncd.
c           If needed, the user can introduce new flags for each patch
c           in the data structure by an extension of the array PPTY
c           with additional rows.
c
c  DEFAULT SCANNING ORDER
c  NrmOrd   A default ordering that can be used by the subroutine
c          'ScanGr'.  This order is a proper actual value for the
c           parameter 'myord' of 'ScanGr'.  The scanning procedure
c           is a recursive process: from the viewpoint of a parent-patch,
c           the kid-patches are visited in the order XKL, XKR, YKL,
c           YKR, ZKL, ZKR (the default ordering).
c           The user can choose his own favourite ordering by adapting
c           the parameter 'myord' (see section 3.2 of the report).
c
c  Below we describe briefly the meaning of the remaining variables in
c  the labeled COMMON Block /DatGlb/. However, they are meant for
c  internal use within the code and therefore of no deep concern for
c  the common user.
c
c  DYNAMIC USE OF WORKSPACE
c  LstSpa    Points to the first empty space for a new patch to be
c            created.
c  NOP       Number of Patches in use.
c
c  SIZE
c  XSize     Number of cells in the X-direction at zero X-level.
c  YSize     Number of cells in the Y-direction at zero Y-level.
c  ZSize     Number of cells in the Z-direction at zero Z-level.
c
c  AT THE ROOTS
c  RtPtr     The Root Pointer.
c  XRoot     The Root-level in the X-direction.
c  YRoot     The Root-level in the Y-direction.
c  ZRoot     The Root-level in the Z-direction.
c
c  GEOMETRY
c  GeomOK    This flag denotes whether the geometry has already been
c            established on level (0,0,0).
```

```
c
c  XCycl     When this flag is set to .TRUE., it denotes the use of
c            cyclic coordinates in the X-direction.
c  YCycl     When this flag is set to .TRUE., it denotes the use of
c            cyclic coordinates in the Y-direction.
c  ZCycl     When this flag is set to .TRUE., it denotes the use of
c            cyclic coordinates in the Z-direction.
c
c  MISCELLANEA
c  TwoPow    This integer array contains the powers of 2 (after the
c            initialisation by subroutine 'IniBas').
c
c
c  END OF DESCRIPTION OF INCLUDE FILE  + + + + + + + + + + + + + + + + +
c
c  DESCRIPTION OF SUBROUTINES AVAILABLE TO THE USER  + + + + + + + + + +
c
c
c  The data structure is handled by the following subroutines (see
c  also Section 3.1 of the report):
c
c
c  SUBROUTINE TO INITIALIZE THE DATA STRUCTURE
c
c     Subroutine IniBas
c     - Corresponds to the PASCAL procedure 'InizBasis3' -
c         A subroutine to be called once, at the outset of a run,
c     before the data structure is actually used. This subroutine
c     initialises pointers, properties, scanning order etc.
c
c  SUBROUTINES FOR THE CONSTRUCTION OF A DOMAIN
c
c     Subroutine MkBloc( dimX,dimY,dimZ, xcycli,ycycli,zcycli)
c     integer dimX,dimY,dimZ
c     logical xcycli,ycycli,zcycli
c     - Corresponds to the PASCAL procedure 'MakeBlock' -
c         Firstly, this subroutine calls 'IniBas'. Secondly, it
c     constructs a data structure corresponding to a particular
c     domain. When the input-parameters 'xcycli', 'ycycli' and
c     'zcycli' are all set to .FALSE., then the domain is a rectangular
c     block. When 'xcycli' is set to .TRUE., this creates cyclic
c     coordinates in the X-direction (beginning and end of the X-interval
c     are glued together, e.g. of use when applying periodic boundary
c     conditions). The parameters 'ycycli' and 'zcycli' have an
c     analogous meaning for the Y- and Z-direction respectively.
c     The input-parameters 'dimX','dimY','dimZ' denote the number of
c     cells in the X-, Y- and Z-direction. The cells created by MkBloc
c     exist only on the (0,0,0)-level.
c     To create more general domains on the (0,0,0)-level, elementary
c     cells can be removed from the block by subsequent calls to the
c     subroutine 'RmCel'.
c
c
c  SUBROUTINES TO ADD TO OR REMOVE FROM THE DATA STRUCTURE
c
c     Subroutine MkKid(ktyp, daddy)
c     integer ktyp, daddy
c     - Corresponds to the PASCAL procedure 'MakeKid' -
```

```
c         A kid cell is created of type 'ktyp' at the cell
c     corresponding to the 'daddy'-patch.
c     The type 'ktyp' wished for, is singled out from the set (XKL,
c     XKR, YKL, YKR, ZKL, ZKR).
c     The kid cell is created provided that in both the X-, Y- and
c     Z-direction the parent exists (otherwise subroutine 'MkFami'
c     should be used).
c
c     Subroutine MkOfsp(patch)
c     integer patch
c     - Corresponds to the PASCAL procedure 'MakeOffspring' -
c         When a patch is complete and pregnant in the X- and/or
c     Y- and/or Z-direction, the Offspring in the corresponding
c     direction is launched by this subroutine.
c     Example of use:
c     call ScanGr(l,m,n, myord, MkOfsp)
c
c     Subroutine MkFami(l,m,n, i,j,k)
c     integer l,m,n, i,j,k
c     - Corresponds to the PASCAL procedure 'MakeFami' -
c         At level (l,m,n) and the location (i,j,k), a cell is
c     created. Parents, grandparents etc. are created as well when
c     they are not already present. (In this way, the creation of
c     necessary intermediate generations will not accidentally be
c     skipped.)
c
c     Subroutine RmCel(patch)
c     integer patch
c     - Corresponds to the PASCAL procedure 'RemoveCell' -
c         If possible (the cell is complete and hasn't any kids) the
c     cell, corresponding to the patch 'patch', is removed (possibly,
c     the patch remains as an incomplete patch).
c
c     Subroutine RmOfsp(patch)
c     integer patch
c     - Corresponds to the PASCAL procedure 'RemoveOffspring' -
c         When the patch 'patch' is complete and its kids are
c     Sentenced, the Offspring is removed.
c     Example of use:
c     call ScanGr(l,m,n, myord, RmOfsp)
c
c
c  SUBROUTINES TO SCAN THE DATA STRUCTURE
c
c     Subroutine ScanGr(l,m,n, myord, DoIt)
c     integer l,m,n, myord(1:3)
c     external DoIt
c     - Corresponds to the PASCAL procedure 'ScanGrid' -
c         A subroutine that scans all patches on the grid with
c     level (l,m,n). The patches are visited by means of a recursive
c     algorithm in an order steered by 'myord'.
c     At each patch visited, a call is made to the subroutine 'DoIt':
c     call DoIt(patch)
c     where the integer 'patch' identifies the patch visited.
c     The subroutine 'DoIt' can be any subroutine constructed
c     by the user, provided that it has the above syntax.
c     If necessary, additional communication between the
c     actual subroutine 'DoIt' and the (sub)program calling
```

```
c     'ScanGr' can be taken care of by a locally defined labeled
c     COMMON Block, shared by the (sub)program calling 'ScanGr' and
c     the actual 'DoIt'.
c
c
c  SUBROUTINES TO SHOW OR TO DUMP DATA
c
c     Subroutine ShoGr(l,m,n)
c     integer l,m,n
c     - Corresponds to the PASCAL procedure 'ShowGrid' -
c         Of all the patches that occur on level (l,m,n) the
c     properties and pointers are printed in an abbreviated
c     manner. The patches rank in the order determined recursively
c     by 'NrmOrd'.
c
c     Subroutine ShoLv(lev)
c     integer lev
c     - Corresponds to the PASCAL procedure 'ShowLevel' -
c         This subroutine calls 'ShoGr' for all levels (l,m,n)
c     with  l+m+n = lev.
c
c     Subroutine DmpAll(dumpfi, name)
c     integer dumpfi
c     character*8 name
c     - Corresponds to the PASCAL procedure 'DumpAll' -
c         Part of the bulk of the data structure (some of the pointers,
c     all properties and the data represented with a limited number
c     of digits) is dumped on a file with unit specifier 'dumpfi',
c     and 'name' a character expression giving the name of the file.
c     This subroutine may serve as a tool for visualisation.
c
c
c  ADDITIONAL SUBROUTINES
c
c     Subroutine error(p, name, nr)
c     integer p, nr
c     character*(*) name
c     - Corresponds to the PASCAL procedure 'error' -
c         A subroutine that is called after a fatal error in the
c     program has occurred.
c     The string 'name' and the integer 'nr' are printed on
c     standard output. Properties and pointers residing in the
c     patch 'p' are printed in an abbreviated manner.
c
c     Subroutine warnin(p, name, nr)
c     integer p, nr
c     character*(*) name
c     - Corresponds to the PASCAL procedure 'warning' -
c         A subroutine that is called after a non-fatal error in the
c     program has occurred.
c     The string 'name' and the integer 'nr' are printed on
c     standard output. Properties and pointers residing in the
c     patch 'p' are printed in an abbreviated manner.
c
c  END OF DESCRIPTION OF SUBROUTINES AVAILABLE TO THE USER  + + + + + +
c
c  NOTE
c     This code passes the Fortran-checker program FTNCHEK
```

```
c      (R. Moniot et al.) with the options
c      FTNCHEK -declare -f77 -portable -novice=2
c
```