Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

P.W. Hemker, H.T.M. van der Maarel, C.T.H. Everaars

BASIS: A data structure for adaptive multigrid computations

Department of Numerical Mathematics Report NM-R9014 August

BASIS : A Data Structure for Adaptive Multigrid Computations

P. W. Hemker
H.T.M. van der Maarel
C.T.H. Everaars

Centre for Mathematics and Computer Science
P.O. box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

In this report a data structure and basic algorithms are described, that can be used for the implementation of adaptive multigrid procedures. The basic elements of the grid are rectangular cells that fit in a QUAD-tree structure. A PASCAL prototype implementation is given and also a FORTRAN implementation is available.

1987 CR Categories: E.1, E.2
1980 Mathematics Subject Classification: 68P05, 65N50
Keywords & Phrases: adaptive multigrid, data structure
Note: This research was performed as part of a BRITE/EURAM project under Contract no. AERO-0003-C.

1 The geometric structure

1.1 Introduction

In this report we describe a data structure that can be used for the solution of partial differential equations. It is meant for the implementation of multigrid methods and handling the corresponding data on a self-adaptive mesh. In particular, it will be applied in a program for the solution of the Euler- and compressible Navier-Stokes equations. A generalization to three or more dimensions is straightforward, only for notational convenience it is not explicitly described here.

We assume that the domain of definition of the partial differential equations, $\Omega \in \mathbb{R}^2$, satisfies sufficient conditions, in order that -possibly after a suitable transformation- a regular partitioning by quadrangles can be made. The boundary of Ω is denoted by $\partial\Omega$.

1.2 Cells and neighbours

The domain of definition Ω is divided into a regular partitioning of a finite number of quadrangles $\Omega^0_{i,j}$,

$$\Omega = \bigcup_{(i,j)\in I} \Omega^0_{i,j}\,, \tag{1}$$

Report NM-R9014
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

in such a way that all quadrangles have, in each of the directions North, East, South or West, either *none* or *one* neighbour quadrangle $\Omega^0_{i,j+1}$, $\Omega^0_{i+1,j}$, $\Omega^0_{i,j-1}$ or $\Omega^0_{i-1,j}$. This means that for each $(l,m) = (0,1),(1,0),(0,-1),(-1,0)$ the cell $\Omega^0_{i,j}$ shares either no or exactly one complete edge with the quadrangle $\Omega^0_{i+l,j+m}$. The index set I is a finite subset from either $\mathbb{Z}^2$ or $\mathbb{Z}_{N_1} \times \mathbb{Z}_{N_2}$, for some natural numbers N_1 or N_2, so that the domain Ω may be (part of) a cylinder or torus. In this way, for each cell its N, E, S or W- neighbour cell (if it exists) is well determined. For each cell we further can identify its N, E, S and W edge and its NE, SE, SW and NW corner[1]. However, we notice that an edge can have positive or zero length. This allows the possibility of cells degenerating to triangles, or even cells of which the area vanishes.

The set $L^0 = \{\Omega^0_{i,j} | \Omega^0_{i,j} \subset \Omega\}$ is the set of cells on level zero. It exactly covers the domain Ω.

Remark 1.1 *The structure of* Ω *and its actual subdivision is not changed if the cells are renumbered in such a way that a constant integer vector* (c_1, c_2) *is added to the index vector* (i,j). *Therefore, without loss of generality we can number the cells so that the lowest indices are zero, so that* $\min\{i|\Omega^0_{i,j} \subset \Omega\} = 0$, *and* $\min\{j|\Omega^0_{i,j} \subset \Omega\} = 0$.

A cell $\Omega^k_{i,j}$ on level k may be divided into 4 disjoint cells on level $k+1$,

$$\Omega^k_{i,j} = \bigcup_{l,m=0,1} \Omega^{k+1}_{2i+l,2j+m}. \tag{2}$$

Remark 1.2 *Cells on levels* $k > 0$ *always appear in quadruples.*

Remark 1.3 *The division in cells on finer levels (i.e. levels with a larger k) are always made in such a way that two cells share either no edge at all, or they share a complete edge.*

Remark 1.4 *The division is made such that in the limit for* $k \to \infty$, *the length of the largest edge on level k vanishes, i.e.* $\lim_{k\to\infty} h_k = 0$.

Definition 1.5 (level k) *The set* $L^k = \{\Omega^k_{i,j} | \Omega^k_{i,j} \subset \Omega\}$ *is the set of cells on level k. The number of cells in* L^k *is denoted by* N_k. *(For* $k > 0$, N_k *satisfies* $N_k \leq 4N_{k-1}$.*)*

Definition 1.6 ((ξ,η)-coordinates) *We provide* Ω *with a* (ξ,η)*-coordinate system such that for the South-West corner of each cell* $\Omega^k_{i,j}$ *the coordinates are given by*

$$(\xi,\eta) = (i\ 2^{-k}, j\ 2^{-k}). \tag{3}$$

Remark 1.7 *It is easy to see that the construction of such a coordinate system is possible and that each point in* Ω *is determined by a pair of coordinates. This coordinate system has little relation with the coordinate system(s) in possible applications. The present* (ξ,η) *coordinate system determines the topological structure of the domain.*

Definition 1.8 (neighbours) *Two cells* $\Omega^k_{i,j}$ *and* $\Omega^n_{l,m}$ *are called neighbours if* $k = n$ *and* $|i-l| + |j-m| = 1$.

Remark 1.9 *Each cell can have 0,1,2,3 or 4 neighbours.*

Definition 1.10 (wall) *An edge of a cell is also called wall.*

---

[1] In this report the orientation of the winds, as well as the meaning of the words 'left', 'right', 'bottom' and 'top' are related in the usual way with the (i,j)- indices.

Remark 1.11 *Because two cells either share no edge at all or a complete edge, each edge has either 1 or 2 neighbouring cells.*

Definition 1.12 (V-type, H-type) *A wall between two cells* $\Omega^k_{i,j}$ *and* $\Omega^k_{i+1,j}$ *is called of V-type; a wall between two cells* $\Omega^k_{i,j}$ *and* $\Omega^k_{i,j+1}$ *is called of H-type.*

Definition 1.13 (corner points) *End points of walls are corner points.*

By the regular partitioning in cells, at each corner point at most 4 cells meet. This is easily seen by means of the (ξ,η)-coördinate system.

1.3 Ghost cells, parents and kids

In order to form a QUAD tree of cells with a single root, we have to combine cells on level 0 in groups on level -1, and recursively, groups on level k in groups on level k-1, until they are all combined in a single group that encloses them all. Therefore we introduce 'ghost cells' (i.e. groups of cells) in the following way. Notice that ghost cells are *not* cells!

Definition 1.14 (ghost cell) *A ghost cell* $\Omega^{-k}_{i,j}$*,* $k > 0$*, is*

$$\Omega^{-k}_{i,j} = \bigcup_{l,m=0,1} \Omega^{-k+1}_{2i+l,2j+m}. \tag{4}$$

Definition 1.15 (enclosing level) *The smallest* $-k$ *for which only one ghost cell (or cell) exists is called the enclosing level,* k_1*.*

Remark 1.16 *The enclosing level* k_1 *satisfies* $k_1 \leq 0$*.*

Definition 1.17 (enclosing cell) *The (ghost) cell on the enclosing level is called the enclosing cell.*

Remark 1.18 *By the choice of the indices* (i,j) *on the level* $k = 0$*, such that the left-most boundary fits with* $i = 0$ *and the bottom with* $j = 0$*, it is clear that the enclosing cell is identified by* $\Omega^{k_1}_{0,0} = \Omega$*.*

Definition 1.19 (parent, kid cell) *For a family of (ghost) cells on the level* k *and* $k+1$*, related by*

$$\Omega^k_{i,j} = \bigcup_{l,m=0,1} \Omega^{k+1}_{2i+l,2j+m}, \qquad k \geq k_1, \tag{5}$$

the (ghost) cell $\Omega^k_{i,j}$ *is called the parent and* $\Omega^{k+1}_{2i+l,2j+m}$ *are called the kids.*

Definition 1.20 (internal, boundary, green wall) *A wall that has two neighbouring cells is called an internal wall. If it has only one neighbour cell, it is called a boundary wall if it is part of* $\partial\Omega$*, or it is called a green wall if it is not part of* $\partial\Omega$*.*

Definition 1.21 (internal, boundary, green point) *If a corner point is the endpoint of a boundary wall, it is called a boundary point. If it is the endpoint of a green wall, it is called a green point. If it isn't a boundary point or green point, it is called an internal point.*

Remark 1.22 *A corner point can be both a boundary and a green point at the same time. Notice that 4 cells and 4 walls meet at a corner point that is an internal point,*

Definition 1.23 (internal, boundary, green cell) *If a cell has a boundary wall, and hence a boundary point as a corner point, it is called a boundary cell. Similarly a cell is a green cell if it has a green wall. If a cell is not a boundary or green cell, it is an internal cell.*

Summary

- There is one enclosing cell that exacly covers Ω;
- A ghost cell lives on negative levels and can have 1,2,3 or 4 kids;
- A cell lives on non-negative levels and can only have 0 or 4 kids;
- We don't associate walls and corner points with ghost cells. (No walls or corner points are defined for negative levels);
- A cell has 4 walls (N,E,S or W), and 4 corner points (NE, SE, SW or NW) that possibly coincide;
- A wall has 1 or 2 neighbour cells;
- A wall is either of H-type or of V-type;
- A corner point can have 1,2,3 or 4 neighbour cells (in any combination, i.e. 15 possible relative locations);
- On level $k = 0$ a cell can have 0,1,2,3 or 4 neighbours; on level $k > 0$ a cell has 2,3 or 4 neighbours;
- A corner point can have 2,3 or 4 neighbour corner points;
- Two cells meet at an interior wall;
- Non-interior walls are either boundary or green walls;
- Four cells meet at an interior point;
- Non-interior points can be boundary and/or green points;
- Green points and walls can only exist for levels $k > 0$.

1.4 The geometric system

Here we give a numbering system for walls and corner points. Of course, walls and corner points are identified by their level and their (ξ,η)-coordinates. For an easier reference, and in order to introduce a numbering system that can be used in a computer implementation, we assume the following convention:

$P^k_{i,j}$ is the corner point on level k, with coordinates $P^k_{i,j} = (i\ 2^{-k}, j\ 2^{-k})$;
$\Gamma^k_{V,i,j}$ is the wall on level k, with coordinates $\{(i\ 2^{-k}, t\ 2^{-k}) | j \leq t \leq j+1\}$;
$\Gamma^k_{H,i,j}$ is the wall on level k, with coordinates $\{(t\ 2^{-k}, j\ 2^{-k}) | i \leq t \leq i+1\}$.
All items $P^k_{i,j}$, $\Gamma^k_{V,i,j}$, $\Gamma^k_{H,i,j}$ and $\Omega^k_{i,j}$ that exist, together form the *geometric system.*

By the selection of the SW-corner in the definition 1.6 we have introduced an asymmetry in the data structure, in the sense that the point $(i\ 2^{-k}, j\ 2^{-k})$ is not the centre of the cell $\Omega^k_{i,j}$. This asymmetry could easily be removed by introducing half-indices $P^k_{i-1/2,j-1/2}$ or $\Omega^k_{i+1/2,j+1/2}$. However, because it is our intention to end up with an actual implementation for which indices better can be successive integers, and data in an array can better be of the same type, we accept the minor inconvenience of the asymmetry.

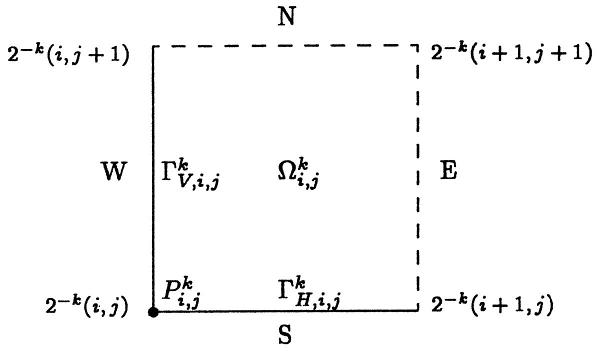

Figure 1: The relative location of the cell, walls and point in the patch $\Pi^k_{i,j}$.

1.5 The patch

Definition 1.24 (patch) *To each corner point $P^k_{i,j}$ in the geometric system we associate a patch of data*

$$\Pi^k_{i,j} = (P^k_{i,j}, \Gamma^k_{V,i,j}, \Gamma^k_{H,i,j}, \Omega^k_{i,j}) \ , \ k \geq 0, \tag{6}$$

where possibly $\Gamma^k_{V,i,j}$, $\Gamma^k_{H,i,j}$ and $\Omega^k_{i,j}$ are empty if they do not exist in the geometric system.

Remark 1.25 *Because each cell has a single SW corner point and each wall has either a single S or W end point, all cells and all walls can be found in exactly one patch.*

Remark 1.26 *Because each cell has 4 corner points, the complete description of a cell - including its walls and corner points- always needs 4 patches.*

Definition 1.27 (parent patch) *With any non-empty set of patches $\{\Pi^{k+1}_{2i+l,2j+m}; l,m = 0,1\}$ we associate its parent patch $\Pi^k_{i,j}$.*

Definition 1.28 (kid patch) *Patches that share the same parent $\Pi^k_{i,j}$ are called the kid-patches of this parent.*

Remark 1.29 *By the definitions it follows immediately that if a parent-kid relation exists between two (ghost) cells $\Omega^k_{i,j}$ and $\Omega^{k+1}_{2i+l,2j+m}$, then the same relation exists for the patches $\Pi^k_{i,j}$ and $\Pi^{k+1}_{2i+l,2j+m}$.*

Definition 1.30 (root level) *The smallest k for which only* one *patch exists is called the root level, k_0.*

Definition 1.31 (root patch) *The patch on the root level is called the root patch.*

Remark 1.32 *The root patch is $\Pi^{k_0}_{0,0}$; the root level k_0 satisfies $k_0 = k_1 - 1 < 0$.*

Definition 1.33 (ghost patch) *The patches $\Pi^k_{i,j}$ with $k_0 \leq k < 0$ are called ghost patches*

$$\Pi^k_{i,j} = (\emptyset, \emptyset, \emptyset, \Omega^k_{i,j}) \ , \ k < 0, \tag{7}$$

where possibly $\Omega^k_{i,j}$ is empty.

Remark 1.34 *Ghost patches are patches!*

Remark 1.35 *Notice that there are more ghost patches than ghost cells.*

Definition 1.36 (complete patch) *If $k \geq 0$ and $\Omega_{i,j}^k$ is* not *empty, then also $\Gamma_{V,i,j}^k$ and $\Gamma_{H,i,j}^k$ are not empty, and the patch is called complete.*

Definition 1.37 (H- or V- patch) *If $k \geq 0$ and $\Gamma_{H,i,j}^k$ (or $\Gamma_{V,i,j}^k$) is* not *empty, then the patch is called a H- or V-patch, respectively.*

Definition 1.38 (thin patch) *If $k \geq 0$ and a patch is not complete, it is called thin. Thin patches can be called V-thin (if $\Gamma_{V,i,j}^k$ exists), and H-thin (if $\Gamma_{H,i,j}^k$ exists).*

Remark 1.39 *Thin patches can be any combination of V- and H-thin; they can be V-, H-, HV-, and 0-thin. The patch is 0-thin if only its corner-point contributes to the geometric system. It is HV-thin in the case of a reentrant corner.*

Remark 1.40 *By construction, it is clear that for all positive levels, a cell or patch has all kids available if and only if its NE-kid exists.*

Definition 1.41 (neighbour patch) *For each patch $\Pi_{i,j}^k$ we define its neighbours $\Pi_{l,m}^k$ by the requirement $|i-l|+|j-m|=1$. Similar to the cells we can identify patches as N-, E-, S- or W-neighbours.*

Summary

- There exists exactly one root patch on the root level k_0;
- There exist ghost patches ($k_0 \leq k < 0$); possibly they contain a ghost cell; they contain no walls or corner points; they have neighbour and parent-kid relations;
- There exist patches that are complete ($k \geq 0$); they contain a cell, two walls and a corner point;
- There exist patches that are not complete ($k \geq 0$); they contain a corner point, possibly one or two walls and no cell;
- Patches that are complete have ($k \geq 0$): 1 parent; 0,1,2,3 or 4 kids; 0,1,2,3 or 4 neighbours; complete data contents;
- Patches that are thin (not complete) have ($k \geq 0$): 1 parent; 0,2 or 3 kids; 1,2,3 or 4 neighbours; partial data contents.

1.6 Conclusion

The patches, introduced in section 1.5, together with the parent-kid relations form a tree. The root patch is found on level k_0. The patches on the levels $k \geq 0$ are all associated with corner points in the geometric structure.

The subset of patches $\Pi_{i,j}^k$, for which there exists a (ghost) cell $\Omega_{i,j}^k$ in the geometric structure, forms a genuine sub-tree in the tree of patches. The patches in this sub-tree that are on levels $k \geq 0$, are the complete patches, which contain the cells in the geometric structure.

2 The data structure

The data structure is ordered by patches. Hence it forms a tree with at most 4 new branches at each node: the QUAD tree. This is the basic structure in which all data about the problem are stored. In the following (sub)sections we describe how the tree is implemented and how operations on it can be executed. One of the aims of the data structure is to provide the means to program multigrid algorithms for adaptive mesh refinement. These algorithms can be based on cell centered schemes as well as based on vertex centered schemes.

With each patch we also associate some *pointers*, *properties*, *coordinates* and *data contents*, of which different parts are associated with the corner point, V-wall, H-wall and/or the cell in the patch. The data contents is any kind of information in the form of real or integer numbers etc. related with the elements of the geometric system. Each patch, except the root patch, contains a pointer to its parent, and all patches contain pointers to their existing kids. Further, each patch has pointers to all its existing neighbour patches. To indicate the non-existence of kids or neighbours a *nil pointer* is used.

2.1 The implementation

Although PASCAL is really well suited to properly implement the data structure that is described in this report, for some trivial but practical reasons it can be decided that it should be implemented (also) in FORTRAN.

The 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 essential restrictions that are inherent to the use of FORTRAN. The useful features as pointers and recursive procedures, that are available in PASCAL but not in FORTRAN, were abandoned in this prototype.

Because the data structure is organized by patch, each patch is given its unique natural number. The number of the root patch is 1. The other patches are sequentially ordered, in order of appearance. Only to accommodate the implementation, we have to specify the maximum number of patches 'MaxNumberOfPatches'.

The pointers in the patch are implemented as integers referring to the corresponding (parent-, neighbour- or kid-) patch. Properties of patches are implemented in a Boolean (logical) array. The coordinates are given in an integer array. The data contents is (mainly) found in the real or double precision arrays of the data structure.

2.2 Pointers

An integer array 'PNTR', dimensioned (FirstPointer: LastPointer, 0: MaxNumberOfPatches), is used for keeping the pointers. For each patch a set of nine pointers is reserved.

Array element	**the integer referring to**
PNTR(Parent,Patch)	the parent of Patch;
PNTR(NE,Patch)	the NE kid of Patch;
PNTR(SE,Patch)	the SE kid of Patch;
PNTR(SW,Patch)	the SW kid of Patch;
PNTR(NW,Patch)	the NW kid of Patch;
PNTR(N,Patch)	the N neighbour of Patch;
PNTR(E,Patch)	the E neighbour of Patch;
PNTR(S,Patch)	the S neighbour of Patch;
PNTR(W,Patch)	the W neighbour of Patch;

The *nil pointer* is the pointer that refers to a nonexisting patch. The nil pointer is implemented as 0 (zero).

Remark 2.1 *We dimension the array* PNTR *as* (FirstPointer : LastPointer, 0: MaxNumberOfPatches). *For all the patches the information about the pointers is contained in* PNTR (*,1:MaxNumberOfPatches). *The elements of* PNTR(*,0), *the pointers of the* nil patch, *are all initialized (and kept) equal to zero. That makes that any pointer from the nilpatch 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 2.2 *Whereas the order of the pointers has no intrinsic meaning, the actual implementation is made by named integer constants*[2]*. However, we fix the order of the winds so that the ordering of these eight winds can be used in a loop*[3]*.*

The index bounds are given by FirstPointer$\leq$0 and LastPointer$\geq$0. These values can be changed in order to accommodate the storage of the integer data if necessary. In fact, in the FORTRAN implementation we will have FirstPointer$\leq$-3, because the integer array Location(0:2,*), as described in section 2.4, is actually stored as Location(i,j) = PNTR(i-3,j) (cf. section 3.4).

2.3 Properties

Various properties of a patch will be available in a Boolean array. 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 available to check the consistency of the data, to report and, eventually, to correct the properties.) Some other (additional) information in the form of properties, can be defined by the user.

Properties of a *patch* that are of interest are the following: complete, thin (HV, H, V or 0). Properties of the corresponding (ghost) *cell* are: ghost cell or genuine cell; boundary cell, green cell or internal cell; all (NE, SE, SW and NW) kids available. Properties of the H- or V-*wall* are: boundary wall, internal wall, green wall. Information that is provided by the user can e.g. be the type of boundary condition (Dirichlet, Neumann, etc.).

Notice that for all positive levels, a cell or patch has all kids available if and only if its NE-kid exists (cf. 1.40). Therefore no properties are introduced to denote the existence of kids for a particular cell.

Other properties of interest for the data structure are 'pregnant', 'sentenced', and 'dead'. The property 'pregnant', for a complete patch, is set to *true* if the corresponding cell is to be refined at the next occasion. Similarly 'sentenced' is used to denote that the cell is to be deleted at the next occasion. A 'dead' patch is the space that remains when a patch has been removed from the data structure. This space contains only garbage to which no pointer is referring. At the next creation of a new patch, this space can be reused.

Some of the above property data are stored in the Boolean (logical) array PPTY, dimensioned (FirstPpty:LastPpty, 1:MaxNumberOfPatches)

Array element[4]	**referring to**
PPTY(Complete, Patch)	Complete patch;
PPTY(WallH, Patch)	$\Gamma_{H,i,j} \in \Pi_{i,j}$
PPTY(WallV, Patch)	$\Gamma_{V,i,j} \in \Pi_{i,j}$

---

[2] FirstPointer$\leq$0, LastPointer$\geq$8, Parent=0, NE=1, SE=2, SW=3, NW=4, N=5, E=6, S=7, W=8)

[3] Although the (main) purpose of the use of named integer constants is that the number of pointers etc. can be extended and that the order can be shuffled, under no circumstances the numbering of the winds will be changed!

PPTY(BdyPoint, Patch)	$P_{i,j} \in \partial\Omega$
PPTY(BdyWallH, Patch)	$\Gamma_{H,i,j} \in \partial\Omega$
PPTY(BdyWallV, Patch)	$\Gamma_{V,i,j} \in \partial\Omega$
PPTY(BdyCell, Patch)	$\Omega_{i,j}$ touches the boundary;
PPTY(GrnPoint, Patch)	$P_{i,j}$ is endpoint of a green wall;
PPTY(GrnWallH, Patch)	$\Gamma_{H,i,j}$ is a green wall;
PPTY(GrnWallV, Patch)	$\Gamma_{V,i,j}$ is a green wall;
PPTY(GrnCell, Patch)	$\Omega_{i,j}$ touches a green wall;
PPTY(Pregnant, Patch)	to be refined at next occasion;
PPTY(Sentenced, Patch)	to be deleted at next occasion;
PPTY(Dead, Patch)	a garbage patch, to be reused;

Remark 2.3 *Whereas the order of the Boolean in the array has no intrinsic meaning, the actual implementation is made by named integer constants*[4].

Remark 2.4 *Another possibility could be to implement the properties in a* character array(FirstPpty:LastPpty, 1:MaxNumberOfPatches), *or in a character string array:* character*14 (1:MaxNumberOfPatches). *The advantages and disadvantages are not yet completely clear (convenience, run-time speed, storage requirements). An advantage might be that a selection from more than two possibilities can be made. E.g. by a character array one could mark a cell as complete, thin or ghost; whereas by a Boolean we only distinguish between complete or not complete.*

Remark 2.5 *The local situation, e.g. the location with respect to a neighbouring boundary, can directly be derived from the properties stored in PPTY and in PNTR. To check the consistency of these data with the actual pointer data the routine CHKPPT can be available.*

2.4 Coordinates

The geometric data of the structure, in the form of (ξ,η)-coordinates, are stored in an integer array Location (0:2, 1:MaxNumberOfPatches), where Location(0,Patch) gives the level k of the patch $\Pi^k_{i,j}$, and Location(1,Patch) and Location(2,Patch) yield the integers i and j respectively.

Generally, the real (physical or (x,y)-) coordinate system will be quite different from the(ξ,η)-coordinate system. To have the real coordinates available, we can proceed in three ways. (1) The real coordinates are made available in the form of a routine that implements the mapping $(\xi,\eta) \to (x,y)$; (2) a real data array is made available COORD (1:2, 1:MaxNumberOfPatches) in which the real coordinates can be found for each point in the structure; or (3) the real coordinates are found in an unformatted direct accessible file (if these data are not frequently accessed).

2.5 Data Contents

The data contents of the data structure is contained in the REAL or DOUBLE PRECISION arrays dimensioned as

DCELL (1:LastCellData, 1:MaxNumberOfPatches)
DHWALL (1:LastWallData, 1:MaxNumberOfPatches)

---

[4] FirstPpty $\leq$1, LastPpty $\geq$14, Complete =1, WallH =2, WallV =3, BdyPoint =4, BdyWallH =5, BdyWallV =6, BdyCell =7, GrnPoint =8, GrnWallH =9, GrnWallV =10, GrnCell =11, Pregnant =12, Sentenced =13, Dead =14.

DVWALL (1:LastWallData, 1:MaxNumberOfPatches)
DPOINT (1:LastPointData, 1:MaxNumberOfPatches)

where e.g. DHWALL(Length[5] ,Patch) gives the length of the horizontal wall in the patch 'Patch'; 'length' being one of the WallData.

Possible 'PointData' are e.g. the function values of the unknown (cell vertex algorithms). Possible 'WallData' are e.g. the length, sine- and cosine of the normal vector on the wall. Possible 'CellData' are e.g the function values of the unknown (cell centered algorithms) and/or the residual.

Remark 2.6 *The user can add his own additional arrays as additional storage for data in the structure, provided that it is dimensioned '1:MaxNumberOfPatches'.*

2.6 Construction of the data structure

For the construction and the handling of the data structure, the following routines should be available to the user. Notice that routines are provided 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 by the newly generated parts.

- **routine** GetRectangle(n,m). Creates the data structure for the domain Ω, where Ω is (topologically equivalent with) a rectangle. At level 0, Ω is a rectangle with $m \times n$ cells.
- **routine** GetCylinder(n,m). Is similar to GetRectangle. However, the domain Ω at level 0 is a cylinder with $m \times n$ cells. The cylinder is located such that the N-side of a rectangle is identified with the S-side. That means that the cylinder has V-walls but no H-walls.
- **routine** Offspring(Ptr). Creates the 4 kids for a complete patch, that is identified by Ptr, and that has no kids yet. It makes an update of the data structure, so that it is consistent with the new status.
- **routine** Educate(Ptr). Fills the data contents of the kids of Ptr, as can be derived from its own contents. This procedure is dependent on the application and should be provided by the user.
- **routine** RemoveOffspring(Ptr). Removes the four kids of the cell identified by Ptr. This routine can be used to remove refinements that are not longer necessary.
- **routine** KillCell(Ptr). Removes the cell identified by Ptr. This routine can be used to re-shape a rectangle or cylinder on level 0 into an arbitrarily shaped region.

Some of the routines (mentioned below) are only auxiliary for the other routines and are not of immediate use to the user.

- **integer function** NewKid. Creates a new patch as one of the four kids of a parent patch. Auxiliary routine.
- **integer function** MakePatch. Creates a new patch and delivers its pointer. Auxiliary routine.
- **routine** RemovePatch(Ptr). If possible (i.e. if neighbour cells don't need this patch for its corner point), this routine removes the patch identified by Ptr from the data structure. Auxiliary routine. (After a successful removal of the patch it is labelled as reusable space by the property *Dead.*)

---

[5] Length is a constant, $1 \leq$ Length $\leq$ LastWallData .

3 The actions on the data structure

3.1 Scanning patches

Because the implementation in FORTRAN prevents the recursive treatment of essentially recursive algorithms, we provide here the non-recursive versions, in order to document the necessary logical spaghetti.

In the following pseudo-code we describe how by a non-recursive algorithm the QUAD tree is searched. This search can be made e.g. for all or part of the patches or cells in the tree. Therefore, besides the root pointer of the (sub-) tree to be scanned (RootPointer), we also specify the lowest (FromLevel, k_a) and highest (ToLevel, k_b) level of the patches $\{\Pi^k_{i,j} | k_a \leq k \leq k_b\}$ to be visited. The order in which the patches are scanned is given by the parameter 'Order'. This is an integer array Order(1:4), which contains a permutation of the directions NE, SE, SW, NW.

```
routine Scan (RootPointer, Order, FromLevel, ToLevel, DoIt );
begin
  if RootPointer $\neq$ NilPointer then
    LowLevel:= LevelOf(RootPointer)
    IPTR(LowLevel):= RootPointer
    if LowLevel = FromLevel then DoIt(RootPointer) endif

    if ToLevel > LowLevel then
      LowLevel + := 1
      for lev from LowLevel to ToLevel do i(lev):= 0 enddo
      lev:= LowLevel

      repeat
        i(lev) + := 1
        if i(lev) $\leq$ 4 then
          IPTR(lev):= PNTR(Order(i(lev)),IPTR(lev-1))
          if IPTR(lev) $\neq$ NilPointer then
            if lev < ToLevel then
              if lev $\geq$ FromLevel then DoIt(IPTR(lev)) endif
              lev + := 1
            else
              DoIt(IPTR(lev))
            endif
          endif
        else if lev > LowLevel then
          i(lev):= 0
          lev - := 1
        endif
      until i(LowLevel) > 4
    endif
  endif
end
```

In fact, the routine SCAN scans all patches. The routine scans the existing patches in the specified order, and visiting each patch, it makes a call to the procedure DoIt(PointerToPatch). The actual routine for 'DoIt' can operate on the patch.

We might need the following routines, that scan patches (cells, walls, H-type walls, V-type walls or points). However, by means of the properties given in PPTY, the implementation of these routines is trivial.

ScanPatches (RootPointer, Order, FromLevel, ToLevel, DoIt)
ScanCells (RootPointer, Order, FromLevel, ToLevel, DoIt)
ScanWalls (RootPointer, Order, FromLevel, ToLevel, DoIt)
ScanHWalls (RootPointer, Order, FromLevel, ToLevel, DoIt)
ScanVWalls (RootPointer, Order, FromLevel, ToLevel, DoIt)
ScanPoints (RootPointer, Order, FromLevel, ToLevel, DoIt) [6]

3.2 The FMG routine

Here we describe in pseudo-code the structure of the global multigrid routine FMG. The integer array nfas(bottomlevel:maxlevel) contains part of the multigrid strategy.

```
routine Fmg (maxlevel,nfas,npmg,nqmg,ncycl,f)
begin
    for toplevel from bottomlevel to maxlevel do
        for i from 1 to nfas(i) do Fas(toplevel) enddo
        $q_h$(toplevel+1) := $P^2_{hH}$(toplevel)
    enddo
end
```

3.3 The FAS cycle

In this section we describe in non-recursive pseudo-code the structure of the nonlinear multigrid routine FAS.

Given are the strategy parameter arrays npmg (bottomlevel:toplevel), ncycl (bottomlevel:toplevel) and nqmg (bottomlevel:toplevel). For each level, from bottomlevel to toplevel, these arrays determine the number of pre- or post-relaxation cycles in each FAS-cycle (in npmg and nqmg respectively). In array ncycl the number of coarse grid correction cycles is given. These are positive integers (non-zero!); ncycl(i)=1 gives a V-cycle, ncycl(i)=2 gives a W-cycle). Also an additional integer f denotes the use of an F-cycle: if f = 1 an F-cycle will be made, otherwise f = 0. Notice that V-F-cycles and W-F-cycles are possible!

```
routine Fas (bottomlevel, toplevel, npmg, nqmg, ncycl, f)
begin
    for i from bottomlevel to toplevel do iters(i):= f enddo
    iters(toplevel) := 1
    level := toplevel

    while iters (toplevel) > 0 do
        while level > bottomlevel do
            PRE-RELAXATION
            for i from 1 to npmg(level) do relax(level) enddo
            RESTRICTION
            $r_H := R_{Hh}(r_h - N_h q_h)$
            $r_H := r_H + N_H q_H$
            $q_H^0 := q_H$
```

---

[6] These are probably less urgent, because, on a positive level, points and patches are similar for this purpose.

```
            level − := 1
            iters(level)+:= ncycl(level)
        enddo

        SOLVE ON COARSEST GRID
        (Here always level = bottomlevel)
        for i from 1 to iters(level) do solve (level) enddo
        iters(level):= 0

        while (level < toplevel) and (iters(level) = 0) do
            PROLONGATION
            $q_h := q_h + P_{hH}(q_H - q_H^0)$
            level + := 1
            POST-RELAXATION
            for i from 1 to nqmg(level) do relax(level) enddo
            iters(level) − := 1
        enddo
    enddo
end
```

3.4 The FORTRAN implementation

In the Fortran implementation all data about the data structure are collected in three ARRAYs and a COMMON BLOCK. The arrays that are declared in the main program are: the integer array PNTR, dimensioned PNTR(FstPtr:LstPtr, 0:MNOP), the logical array PPTY (FstPpt:LstPpt, 0:MNOP); and the real array DATA (1:MNOD, 0:MNOP)[7]. 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. In the actual implementation it is made particularly simple to adapt the parameters MNOD and MNOP. A small number of additional integers is necessary to handle the data structure. These numbers are needed as global variables. They are collected in the common block /DatGlb/ RtLv, LstSpa, NOP, SizeX0, SizeY0, NrmOrd(4)[8].

To reduce the number of arrays that should be available, in the FORTRAN implementation, the integer array Location is also stored into the integer array PNTR, such that Location(i,j) is found in PNTR([i-3,j), (i=0,1,2). The real arrays DCELL, DHWALL, DVWALL and DPOINT are all collected in the single real array DATA, of which the number of rows (MNOD) is to be determined by the user.

3.5 The PASCAL prototype

All essential algorithms with respect to the pointers and the properties in the data structure have been implemented in the PASCAL prototype that is given in this section.

In the main program of this prototype an example of their use has been given. An irregular domain is created. The coarsest grid is refined and, further, this fine grid is partly un-refined and other parts are refined again. The resulting data about the data structure are written to three files 'cel.cp<i>', i=1,2,3, and the UNIX preprocessor 'grap' is used to draw the corresponding meshes. The resulting meshes are shown in the Figures 2 through 5.

```
program prototype (input,output);
```

---

[7] MNOD: Maximum Number Of Data per patch; MNOP: Maximum Number Of Patches.

[8] RtLv = RootLevel; LstSpa = Last Space; NOP = Number Of Patches; (SizeX0, SizeY0) denotes the number of cells on the zero level; NrmOrd = Normal Ordering.

```
   { -- created: November 1989   }
   { -- this version: 1990-5-22  }
   { -- author: P.W. Hemker }

const
   MNOP = 2000;   { -- MaxNumberOfPatches   }
   MNOL =   20;   { -- MaxNumberOfLevels    }
   LNOL =  -10;   { -- LowestNumberOfLevels }
   nihil   = 0;   { -- the nil pointer      }
   RootPointer  = 1;

   FirstPointer = 0; LastPointer =  8;
   Parent=0; NE=1; SE=2; SW=3; NW=4; N=5; E=6; S=7; W=8;

   FirstPpty =  1; LastPpty  = 14;
   Complete  =  1; WallH     =  2; WallV    =  3;
   BdyPoint  =  4; BdyWallH  =  5; BdyWallV =  6; BdyCell =  7;
   GrnPoint  =  8; GrnWallH  =  9; GrnWallV = 10; GrnCell = 11;
   Pregnant  = 12; Sentenced = 13; Dead     = 14;

type
   pointer = integer;
   order   = array[1.. 4]of integer;
   string  = array[1..24]of char;

var
   RootLevel, LastSpace, NumberOfPatches,
   RectSizeX, RectSizeY :integer;
   NormalOrder          :order;

   PNTR      :array [FirstPointer..LastPointer,0..MNOP] of pointer;
   PPTY      :array [FirstPpty   ..LastPpty   ,0..MNOP] of boolean;
   Location  :array [0           ..2          ,1..MNOP] of integer;

procedure error (i :pointer; s1, s2: string);
begin { -- hard error message }
      writeln('fatal error'); writeln(i);
      writeln(s1); writeln(s2); writeln;
end;

procedure warning (i :pointer; s1, s2: string);
begin { -- soft error message }
      writeln('warning ',i,' ',s1,' ',s2);
end;

procedure ReportIt (level: integer; filename :string);
var
   k,m   :integer;
   scale :real;
   report:text;
begin
   scale:= 1;
   for k:= RootLevel to level-1 do scale:= scale/2.0;

   rewrite(report,filename);
```

```
   for m:= 1 to NumberOfPatches do
   if Location[0,m] = level then
   begin
      for k:= FirstPpty to LastPpty do
        if PPTY[k,m] then write(report,'T ') else write(report,'F ');
      write(report,m,'  ');
      for k:= 1 to 2 do write(report,Location[k,m]*scale,'  ');
      writeln(report,scale/2);
   end;
end;

procedure Show (patch: pointer);
var
   k :integer;
   str:array[1..2]of char;
begin
   write(patch:3,' @');
   for k:=   0 to  2 do write(' ',Location[k,patch]:3);
   write(';');
   for k:= FirstPointer to LastPointer do write(PNTR[k,patch]:3);
   for k:= FirstPpty to LastPpty do
     begin if PPTY[k,patch] then
       case k of
         1: str:= 'C_'; 2: str:= 'H_'; 3: str:= 'V_';
         4: str:= 'bP'; 5: str:= 'bH'; 6: str:= 'bV'; 7: str:= 'bC';
         8: str:= 'gP'; 9: str:= 'gH';10: str:= 'gV';11: str:= 'gC';
         12:str:= 'Pr';13: str:= 'Sd';14: str:= 'Dd';
      end else str:= '__';
      write(str);
    end;
  writeln;
end;

function TwoPow ( k: integer) :integer;
var   { -- computes 2**k }
   l,p :integer;
begin
   p:= 1;
   if k>0 then for l:= 1 to k do p:= 2*p
   else if k<0 then error(nihil,'negative argument','TwoPow');
   TwoPow:= p;
end;

procedure InizData ;
var  { -- initialization of the data structure }
   i :integer;
begin
   { -- makes a pointer of the nil pointer the nil pointer! }
   for i:= FirstPointer to LastPointer do PNTR[i,0]:= 0;
   for i:= FirstPpty    to LastPpty    do PPTY[i,0]:= false;

   NormalOrder[1]:= 3; NormalOrder[2]:= 4;
   NormalOrder[3]:= 2; NormalOrder[4]:= 1;
   NumberOfPatches:= 0; LastSpace:= 1;
end;
```

```
function MakePatch (k,i,j :integer; daddy :pointer):pointer;
{ -- Takes only care of Location and the parent pointer. -- }
var
   ii :integer;
begin
   LastSpace:= LastSpace-1;
   repeat LastSpace:= LastSpace+1
       until PPTY[Dead,LastSpace] or (LastSpace>NumberOfPatches);
   if LastSpace>NumberOfPatches then NumberOfPatches:= LastSpace;

   if NumberOfPatches > MNOP then
      error(daddy,'too many patches','MakePatch')
   else
      MakePatch:= LastSpace;

   { -- NO properties, NO kids, NO neighbours, a FATHER }
   for ii:= FirstPointer to LastPointer do
       PNTR[ii,LastSpace]:= nihil;
   for ii:= FirstPpty to LastPpty do
       PPTY[ii,LastSpace]:= false;
   PNTR[Parent,LastSpace]:= daddy;
   Location [0,LastSpace]:= k;
   Location [1,LastSpace]:= i;
   Location [2,LastSpace]:= j;
end;

function NewKid (wind :integer; daddy: pointer):integer;
{ -- Takes only care of Location and the parent pointer. -- }
var
   i,ii,j,jj,k :integer;
begin
   if PNTR[wind, daddy] = nihil then
   { -- only if the kid doesn't yet exist ! -- }
begin
   k := Location[0,daddy];
   i := Location[1,daddy];
   j := Location[2,daddy];
   case wind of
      NE : begin ii:= 2*i+1; jj:= 2*j+1 end;
      SE : begin ii:= 2*i+1; jj:= 2*j   end;
      NW : begin ii:= 2*i  ; jj:= 2*j+1 end;
      SW : begin ii:= 2*i  ; jj:= 2*j   end;
   end;

   PNTR[wind, daddy]:= MakePatch(k+1,ii,jj, daddy);
end;
   NewKid:= PNTR[wind, daddy];
end;

procedure GhostTies ( daddy: pointer);
{ -- This routine constructs the pointers between all existing
  -- kid-patches of daddy and their neighbours (the cousins).
  -- It assumes that neighbour relations already exist between
  -- daddy and his neighbours (the existing uncles).
```

```
  -- Routine ment for levelk < 0
}
var
   NEk, SEk, SWk, NWk :pointer; { -- the kids }
   Nc,  Ec,  Sc,  Wc  :pointer; { -- the cousins }
begin
   { -- give all the kids their names }
   NEk:= PNTR[ NE,daddy];
   SEk:= PNTR[ SE,daddy];
   NWk:= PNTR[ NW,daddy];
   SWk:= PNTR[ SW,daddy];

   { -- first construct brother pointers }
   if NEk*SEk>0 then begin PNTR[S,NEk]:= SEk; PNTR[N,SEk]:= NEk end;
   if NWk*SWk>0 then begin PNTR[S,NWk]:= SWk; PNTR[N,SWk]:= NWk end;
   if NEk*NWk>0 then begin PNTR[W,NEk]:= NWk; PNTR[E,NWk]:= NEk end;
   if SEk*SWk>0 then begin PNTR[W,SEk]:= SWk; PNTR[E,SWk]:= SEk end;

   { -- now construct cousin pointers }
   if NEk <> nihil then
   begin
      Nc:= PNTR[ SE, PNTR[ N, daddy]];
      Ec:= PNTR[ NW, PNTR[ E, daddy]];
      if Nc<>0 then begin PNTR[N,NEk]:= Nc; PNTR[S,Nc]:= NEk end
               else warning(NEk,'has no N-wall','GhostTies');
      if Ec<>0 then begin PNTR[E,NEk]:= Ec; PNTR[W,Ec]:= NEk end
               else warning(NEk,'has no E-wall','GhostTies');
   end;

   if SEk <> nihil then
   begin
      Sc:= PNTR[ NE, PNTR[ S, daddy]];
      Ec:= PNTR[ SW, PNTR[ E, daddy]];
      if Sc<>0 then begin PNTR[S,SEk]:= Sc; PNTR[N,Sc]:= SEk end;
      if Ec<>0 then begin PNTR[E,SEk]:= Ec; PNTR[W,Ec]:= SEk end
               else warning(SEk,'has no E-wall','GhostTies');
   end;

   if NWk <> nihil then
   begin
      Nc:= PNTR[ SW, PNTR[ N, daddy]];
      Wc:= PNTR[ NE, PNTR[ W, daddy]];
      if Nc<>0 then begin PNTR[N,NWk]:= Nc; PNTR[S,Nc]:= NWk end
               else warning(NWk,'has no N-wall','GhostTies');
      if Wc<>0 then begin PNTR[W,NWk]:= Wc; PNTR[E,Wc]:= NWk end;
   end;

   if SWk <> nihil then
   begin
      Sc:= PNTR[ NW, PNTR[ S, daddy]];
      Wc:= PNTR[ SE, PNTR[ W, daddy]];
      if Sc<>0 then begin PNTR[S,SWk]:= Sc; PNTR[N,Sc]:= SWk end;
      if Wc<>0 then begin PNTR[W,SWk]:= Wc; PNTR[E,Wc]:= SWk end;
   end;
end;
```

```
procedure FamilyTies ( daddy: pointer);
{ -- This routine constructs the pointers between all existing
  -- kid-patches of daddy and their the cousins at the NE-sides.
  -- The pointers to the other sides (if necessary) already exist!
  -- It assumes that neighbour relations already exist between
  -- daddy and his neighbours (the existing uncles).
  -- Routine ment for levelk >= 0
}
var
   NEk, SEk, SWk, NWk,          { -- the kids     }
   NENc, NEEc, SEEc,NWNc, NEc, { -- far cousins }
   NUncle, EUncle, NEUncle     { -- the uncles  } :pointer;
begin
     if not PPTY[Complete,daddy] then
        error(daddy,'incomplete','FamilyTies')
     else if PNTR[NE,daddy]<>0 then { -- all kids should exist }
begin
   { -- give all the kids their names }
   NEk:= PNTR[ NE,daddy];
   SEk:= PNTR[ SE,daddy];
   NWk:= PNTR[ NW,daddy];
   SWk:= PNTR[ SW,daddy];
   if NEk*SEk*NWk*SWk = 0 then
      error(daddy,'kid missing ','FamilyTies') else
   begin
      { -- first construct brother pointers }
      PNTR[S,NEk]:= SEk; PNTR[N,SEk]:= NEk;
      PNTR[S,NWk]:= SWk; PNTR[N,SWk]:= NWk;
      PNTR[W,NEk]:= NWk; PNTR[E,NWk]:= NEk;
      PNTR[W,SEk]:= SWk; PNTR[E,SWk]:= SEk;

      { -- give the uncles their names }
      NUncle  := PNTR[N, daddy];
      EUncle  := PNTR[E, daddy];
      NEUncle:= PNTR[N,EUncle];
         if PNTR[E,NUncle] <> NEUncle then
         error(daddy,'inconsistent NEcorner','FamilyTies');

      { -- give the cousins their names }
      NENc:= PNTR[ SE, NUncle];
      NEEc:= PNTR[ NW, EUncle];
      SEEc:= PNTR[ SW, EUncle];
      NWNc:= PNTR[ SW, NUncle];
      NEc  := PNTR[ SW, NEUncle];
         if NENc*NEEc*SEEc*NWNc*NEc = 0 then
         error(daddy,'nil neighbours','FamilyTies');

      { -- now construct cousin pointers }
      PNTR[N, NEk]:= NENc; PNTR[S,NENc]:= NEk;
      PNTR[E, NEk]:= NEEc; PNTR[W,NEEc]:= NEk;
      PNTR[E, SEk]:= SEEc; PNTR[W,SEEc]:= SEk;
      PNTR[N, NWk]:= NWNc; PNTR[S,NWNc]:= NWk;

      PNTR[E,NWNc]:= NENc; PNTR[W,NENc]:= NWNc;
```

```
      PNTR[E,NENc]:= NEc ; PNTR[W,NEc ]:= NENc;
      PNTR[N,NEEc]:= NEc ; PNTR[S,NEc ]:= NEEc;
      PNTR[N,SEEc]:= NEEc; PNTR[S,NEEc]:= SEEc;
   end;
end;
end;


procedure BoundaryProperties(patch :pointer);
{ -- Determines either boundary walls and boundary cells
  -- around a complete patch (on level zero), or
  -- it determines the green walls and green cells
  -- around a complete patch (on a positive level)
  -- Boundary ppties are permanent, they are obtained either on
  -- level 0 or by inheritance. Green ppties may change any time. -- }
var
   NEnb, Nnb, Enb, Snb, Wnb, p :pointer;
   XxxPoint, XxxWallH, XxxWallV, XxxCell, i :integer;
begin
   if patch = nihil then {-- skip } else
begin
   if Location[0,patch]<0 then
      error(patch,'No boundaries','BoundaryProperties')
   else if Location[0,patch]=0 then
      begin
         XxxWallH:= BdyWallH; XxxPoint:= BdyPoint;
         XxxWallV:= BdyWallV; XxxCell := BdyCell ;
      end
   else
      begin
         XxxWallH:= GrnWallH; XxxPoint:= GrnPoint;
         XxxWallV:= GrnWallV; XxxCell := GrnCell ;
      end;

   { -- Give some neighbours their name --}
   Nnb := PNTR[N,patch];
   Enb := PNTR[E,patch];
   NEnb:= PNTR[E,  Nnb];
   Snb := PNTR[S,patch]; { -- possibly nihil !! }
   Wnb := PNTR[W,patch]; { -- possibly nihil !! }

   if PPTY[Complete,patch] then
   begin
      { -- First determine the boundary Walls around a cell --}
      PPTY[XxxWallH,patch]:= not PPTY[Complete,Snb];
      PPTY[XxxWallV,patch]:= not PPTY[Complete,Wnb];
      PPTY[XxxWallH,  Nnb]:= not PPTY[Complete,Nnb] ;
      PPTY[XxxWallV,  Enb]:= not PPTY[Complete,Enb] ;

      if PPTY[BdyWallH,patch] then PPTY[GrnWallH,patch]:= false;
      if PPTY[BdyWallV,patch] then PPTY[GrnWallV,patch]:= false;
      if PPTY[BdyWallH,  Nnb] then PPTY[GrnWallH,  Nnb]:= false;
      if PPTY[BdyWallV,  Enb] then PPTY[GrnWallV,  Enb]:= false;

      { -- Secondly, determine the boundary Points around a cell --}
```

```
      PPTY[XxxPoint,patch]:=
                    PPTY[XxxWallH,patch] or PPTY[XxxWallV,patch] or
                    PPTY[XxxWallH,  Wnb] or PPTY[XxxWallV,  Snb] ;
      PPTY[XxxPoint,  Nnb]:=
                    PPTY[XxxWallH,  PNTR[W,Nnb]] or
                    PPTY[XxxWallV,patch] or
                    PPTY[XxxWallH,  Nnb] or PPTY[XxxWallV,  Nnb] ;
      PPTY[XxxPoint,  Enb]:=
                    PPTY[XxxWallV,  PNTR[S,Enb]] or
                    PPTY[XxxWallH,patch] or
                    PPTY[XxxWallH,  Enb] or PPTY[XxxWallV,  Enb] ;
      PPTY[XxxPoint, NEnb]:=
                    PPTY[XxxWallH,  Nnb] or PPTY[XxxWallV,  Enb] or
                    PPTY[XxxWallH, NEnb] or PPTY[XxxWallV, NEnb] ;

      { -- Finally, determine the boundary cells around the patch }
      PPTY[XxxCell ,patch]:=
            PPTY[XxxWallH,patch] or PPTY[XxxWallH,PNTR[N,patch]]
         or PPTY[XxxWallV,patch] or PPTY[XxxWallV,PNTR[E,patch]];
      for i:= N to W do
      begin
         p:= PNTR[i,patch]; if p <> nihil then
            PPTY[XxxCell ,p]:= PPTY[Complete,p] and
                ( PPTY[XxxWallH,p] or PPTY[XxxWallH,PNTR[N,p]]
               or PPTY[XxxWallV,p] or PPTY[XxxWallV,PNTR[E,p]]  );
      end;
   end
   else { -- the patch is NOT complete -- }
   begin
      PPTY[XxxCell ,patch]:= false;
      PPTY[XxxWallH,patch]:= PPTY[Complete,Snb];
      PPTY[XxxWallV,patch]:= PPTY[Complete,Wnb];

      if PPTY[BdyWallH,patch] then PPTY[GrnWallH,patch]:= false;
      if PPTY[BdyWallV,patch] then PPTY[GrnWallV,patch]:= false;

      PPTY[XxxPoint,patch]:= PPTY[XxxWallH,  Wnb] or PPTY[XxxWallV,  Snb] or
                             PPTY[XxxWallH,patch] or PPTY[XxxWallV,patch];
   end;
end;
end;

procedure Offspring (daddy: pointer);
var
   NEk, SEk, SWk, NWk,      { -- the kids    }
   NUncle, EUncle, NEUncle, { -- the uncles  }
   NWN, NEN, NEE, SEE, NEc  { -- the cousins } :pointer;
   w,k :integer;
begin
     if PPTY[Complete,daddy] then
begin
   k := Location[0,daddy];
   if k < 0 then error(daddy,'negative level','Offspring');
   if PNTR[NE,daddy] <> 0 then
           error(daddy,'child already exists','Offspring');
```

```
SWk:= NewKid(SW, daddy);
SEk:= NewKid(SE, daddy);
NWk:= NewKid(NW, daddy);
NEk:= NewKid(NE, daddy);

{ -- give all the uncles their names }
NUncle := PNTR[ N,daddy];
EUncle := PNTR[ E,daddy];
NEUncle:= PNTR[N,EUncle];
      if PNTR[E,NUncle] <> NEUncle then
      error(daddy,'inconsistent NEcorner','Offspring');

{ -- give the cousins their names }
NWN:= NewKid(SW, NUncle);
NEN:= NewKid(SE, NUncle);
NEE:= NewKid(NW, EUncle);
SEE:= NewKid(SW, EUncle);
NEc:= NewKid(SW,NEUncle);

FamilyTies(daddy);

 { -- take care of the major patch properties }
 for w:= NE to NW do
    begin
       PPTY[Complete,PNTR[w,daddy]]:= true;
       PPTY[   WallH,PNTR[w,daddy]]:= true;
       PPTY[   WallV,PNTR[w,daddy]]:= true;
    end;
 PPTY[WallH,NEN]:= true; PPTY[WallH,NWN]:= true;
 PPTY[WallV,NEE]:= true; PPTY[WallV,SEE]:= true;

 { -- take care of the boundary properties }
 if PPTY[BdyWallH, daddy] then
    begin
       PPTY[BdyWallH,SEk]:= true; PPTY[BdyPoint,SEk]:= true;
       PPTY[BdyWallH,SWk]:= true; PPTY[BdyPoint,SEE]:= true;
    end;
 if PPTY[BdyWallV, daddy] then
    begin
       PPTY[BdyWallV,NWk]:= true; PPTY[BdyPoint,NWk]:= true;
       PPTY[BdyWallV,SWk]:= true; PPTY[BdyPoint,NWN]:= true;
    end;
 if PPTY[BdyPoint, daddy] then    PPTY[BdyPoint,SWk]:= true;

 if PPTY[BdyWallH,NUncle] then
    begin
       PPTY[BdyWallH,NWN]:= true; PPTY[BdyPoint,NWN]:= true;
       PPTY[BdyWallH,NEN]:= true; PPTY[BdyPoint,NEN]:= true;
    end;
 if PPTY[BdyWallV,EUncle] then
    begin
       PPTY[BdyWallV,SEE]:= true; PPTY[BdyPoint,SEE]:= true;
       PPTY[BdyWallV,NEE]:= true; PPTY[BdyPoint,NEE]:= true;
    end;
```

```
         if PPTY[BdyCell,daddy] then
            begin
               if PPTY[BdyWallV, daddy] or PPTY[BdyWallH, daddy]
                 then PPTY[BdyCell,SWk]:= true;
               if PPTY[BdyWallV, daddy] or PPTY[BdyWallH,NUncle]
                 then PPTY[BdyCell,NWk]:= true;
               if PPTY[BdyWallH, daddy] or PPTY[BdyWallV,EUncle]
                 then PPTY[BdyCell,SEk]:= true;
               if PPTY[BdyWallV,EUncle] or PPTY[BdyWallH,NUncle]
                 then PPTY[BdyCell,NEk]:= true;
            end;
         if PPTY[BdyPoint,NEUncle] then   PPTY[BdyPoint,NEc]:= true;

         { -- take care of the green properties }
         for w:= NE to NW do BoundaryProperties(PNTR[w,daddy]);

end else
   warning(daddy,'incomplete patch','Offspring');
end;

procedure HardRemovePatch(patch :pointer);
{ -- This routine removes a cell from the system. }
var
   i :integer;
   daddy :pointer;
begin
   { -- Pointers to the patch disappear -- }
   PNTR[S,PNTR[N,patch]]:= nihil;
   PNTR[W,PNTR[E,patch]]:= nihil;
   PNTR[N,PNTR[S,patch]]:= nihil;
   PNTR[E,PNTR[W,patch]]:= nihil;

   { -- The relation with Parent is closed -- }
   daddy:= PNTR[Parent,patch];
   for i:= NE to NW do
       if PNTR[i,daddy]= patch then PNTR[i,daddy]:= nihil;
   if patch<LastSpace then LastSpace:=patch;

   { -- All pointers are removed -- }
   for i:= FirstPointer to LastPointer do PNTR[i,patch]:= nihil;

   { -- All other properties are removed -- }
   for i:= FirstPpty to LastPpty do PPTY[i,patch]:= false;
   PPTY[Dead,patch]:= true;
end;

procedure RemovePatch(patch :pointer);
{ -- If possible, this routine removes a cell from the system.
  -- If a neighbour cell needs this patch for its corner point
  -- it isn't possible. -- }
begin
   if  not ( PPTY[WallH,patch] or PPTY[WallV,patch]
            or PPTY[Complete,PNTR[S,PNTR[W,patch]]]
            or PPTY[Complete,PNTR[W,PNTR[S,patch]]] ) then
```

```
   HardRemovePatch(patch);
end;

procedure KillCell(patch :pointer);
{ -- Let the cell disappear from the system
  -- the patch possibly remains -- }
var
   Nnb, Enb, Snb, Wnb, NEnb :pointer;
begin
   if PPTY[Complete,patch] then
begin
   if PNTR[NE,patch] <> nihil then
      error(patch,'has kids','KillCell');

   { -- Give the neighbours their name }
   Nnb := PNTR[N,patch];
   Enb := PNTR[E,patch];
   NEnb:= PNTR[N,  Enb]; if NEnb = nihil then NEnb:= PNTR[E, Nnb];
   { -- they're possibly nil: }
   Snb := PNTR[S,patch];
   Wnb := PNTR[W,patch];

   if  not PPTY[WallV   ,patch] then PPTY[BdyWallV,patch]:= false; { -- changed}
   if  not PPTY[WallH   ,patch] then PPTY[BdyWallH,patch]:= false; { -- changed}
   { -- Cell disappears }
   PPTY[ Complete,patch]:= false;
   PPTY[Sentenced,patch]:= false;
   PPTY[  BdyCell,patch]:= false;

   { -- Walls possibly disappear}
   PPTY[WallV,patch]:= PPTY[Complete,Wnb];
   PPTY[WallH,patch]:= PPTY[Complete,Snb];
   PPTY[WallV,  Enb]:= PPTY[Complete,Enb];
   PPTY[WallH,  Nnb]:= PPTY[Complete,Nnb];
   PPTY[WallV, NEnb]:= PPTY[Complete,Nnb] or PPTY[Complete,NEnb];
   PPTY[WallH, NEnb]:= PPTY[Complete,Enb] or PPTY[Complete,NEnb];

   { -- True boundary properties }
   PPTY[BdyWallV,patch]:= PPTY[BdyWallV,patch] and PPTY[WallV,patch];
   PPTY[BdyWallH,patch]:= PPTY[BdyWallH,patch] and PPTY[WallH,patch];
   PPTY[BdyWallV,  Enb]:= PPTY[BdyWallV,  Enb] and PPTY[WallV,  Enb];
   PPTY[BdyWallH,  Nnb]:= PPTY[BdyWallH,  Nnb] and PPTY[WallH,  Nnb];
   PPTY[BdyWallV, NEnb]:= PPTY[BdyWallV, NEnb] and PPTY[WallV, NEnb];
   PPTY[BdyWallH, NEnb]:= PPTY[BdyWallH, NEnb] and PPTY[WallH, NEnb];

   RemovePatch(patch); RemovePatch(  Nnb);
   RemovePatch(  Enb); RemovePatch( NEnb);

   BoundaryProperties(patch);
   BoundaryProperties(Nnb);   BoundaryProperties(Enb);
   BoundaryProperties(NEnb);
   BoundaryProperties(Snb);   BoundaryProperties(Wnb);
end;
end;
```

```
procedure RemoveOffspring(patch :pointer);
{ -- If possible this routine removes the 4 kids of daddy and it
  -- adapts the datastructure correspondingly. It is only possible
  -- if all kids are sentenced. -- }
var
   w :integer;
begin
   if PPTY[Sentenced, PNTR[NE,patch]] and
      PPTY[Sentenced, PNTR[SE,patch]] and
      PPTY[Sentenced, PNTR[NW,patch]] and
      PPTY[Sentenced, PNTR[SW,patch]] then
   for w:= NE to NW do KillCell( PNTR[w,patch]);
end;


procedure Scan (RootPointer: pointer; MyOrder :order;
                FromLevel, ToLevel :integer;
                procedure DoIt (patch :pointer) );
var
   lev,LowLevel   :integer;
   IPTR ,i :array [LNOL..MNOL] of integer;
begin
   if RootPointer = nihil then error(RootPointer,'nil pointer    ','Scan');
   if ToLevel < FromLevel then error(RootPointer,'wrong ToLevels ','Scan');
   LowLevel:= Location[0,RootPointer];
   if LowLevel > FromLevel then error(RootPointer,'wrong LowLevel','Scan');

   IPTR[LowLevel]:= RootPointer;
   if LowLevel = FromLevel then DoIt(RootPointer);

   if ToLevel > LowLevel then
   begin
      LowLevel:= LowLevel + 1;
      for lev:= LowLevel to ToLevel do i[lev]:= 0;
      lev:= LowLevel;

      repeat
         i[lev]:= i[lev] + 1;
         if i[lev] <= 4 then
         begin
            IPTR[lev]:= PNTR[MyOrder[i[lev]],IPTR[lev-1]];
            if IPTR[lev] <> nihil then
            begin
               if lev < ToLevel then
               begin
                  if lev >= FromLevel then DoIt(IPTR[lev]);
                  lev:= lev+1
               end
               else
                  DoIt(IPTR[lev]);
            end
         end
         else if lev > LowLevel then
         begin
```

```
            i[lev]:= 0;
            lev:= lev-1;
         end;
      until i[LowLevel] > 4;
   end;
end;

procedure RectCHV ( patch: pointer);
{ -- Auxiliary for GetRectangle -- }
var
   i,j,k,RX, RY :integer;
begin
      k := Location[0,patch];
      i := Location[1,patch];
      j := Location[2,patch];
      RX:= RectSizeX;
      RY:= RectSizeY;
      if k <> 0           then error(patch,'non-zero level','RectCHV');
      if (i>RX) or (j>RY) then error(patch,'improper RECT ','RectCHV');

      if (i<RX) and (j<RY) then PPTY[Complete,patch]:= true;
      if i < RX then PPTY[WallH    ,patch]:= true;
      if j < RY then PPTY[WallV    ,patch]:= true;
end;

procedure RectKid ( patch: pointer);
{ -- Auxiliary for GetRectangle -- }
var
   k,w,z,tt :integer;
begin
      k := Location[0,patch];
      if k>=0 then error(patch,'positive level','RectKid');
      tt:= TwoPow(-1-k);

      for w:= 1 to 4 do
      begin
        z:= NewKid(NormalOrder[w],patch);
        if (tt*Location[1,z] > RectSizeX) or
           (tt*Location[2,z] > RectSizeY) then RemovePatch(z);
      end;
end;


procedure  GetRectangle ( m,n: integer);
var
   mm, nn, k   :integer;
   RootPointer :pointer;
begin
   RectSizeX:= n;
   RectSizeY:= m;
   if (n<=0) or (m<=0) then
      error(nihil,'invalid input','GetRectangle');
   if n>m then mm:=n else mm:= m;
   RootLevel:= 0; nn:= 1;
   repeat
```

```
         RootLevel := RootLevel-1;
         nn:= nn*2;
   until nn > mm;

   RootPointer:= MakePatch(RootLevel,0,0,nihil);
   for k:= RootLevel to -1 do
           Scan(RootPointer,NormalOrder,k,k,RectKid);
   { -- construct all pointers --}
   for k:= RootLevel to -1 do
           Scan(RootPointer,NormalOrder,k,k,GhostTies);
   { -- construct all CHV properties --}
   Scan(RootPointer,NormalOrder,0,0,RectCHV);
   { -- construct all Bdy properties --}
   Scan(RootPointer,NormalOrder,0,0,BoundaryProperties);
end;

procedure  GetCylinder ( m,n: integer);
var
   Spointer, Npointer, Victim :pointer;
   i  :integer;
begin
   GetRectangle(m,n);
   Spointer:= RootPointer;
   while Location[0,Spointer] < 0 do
      Spointer:= PNTR[SW,Spointer];
   Npointer:= Spointer;
   while PNTR[N,Npointer] <> nihil do
      Npointer:= PNTR[N,Npointer];
   Victim  := Npointer;
   Npointer:= PNTR[S,Victim];
   PPTY[Dead,Victim]:= true;
   HardRemovePatch(Victim);

   for i:= 1 to RectSizeX do
   begin
      Victim:= PNTR[N,PNTR[E,Npointer]];
      PPTY[Dead,Victim]:= true;      HardRemovePatch(Victim);

      PNTR[S,Spointer]:= Npointer;  PNTR[N,Npointer]:= Spointer;
      BoundaryProperties(Npointer); BoundaryProperties(Spointer);
      Npointer:= PNTR[E,Npointer];  Spointer:= PNTR[E,Spointer];
   end;
   PNTR[S,Spointer]:= Npointer;  PNTR[N,Npointer]:= Spointer;
   BoundaryProperties(Npointer); BoundaryProperties(Spointer);
end;

procedure CHKCIRC (patch :pointer);
    { --This routine checks some consistency
      -- of the neighbour pointers.
    }
var
   NW1,NW2,NE1,NE2, SW1,SW2,SE1,SE2 :pointer;
begin
   NW1:= PNTR[ N, PNTR[ W, patch]];
   NW2:= PNTR[ W, PNTR[ N, patch]];
```

```
   if NW1 <> NW2 then writeln('CHKCIRC',patch,'   ',NW1,'   ',NW2);

   SW1:= PNTR[ S, PNTR[ W, patch]];
   SW2:= PNTR[ W, PNTR[ S, patch]];
   if SW1 <> SW2 then writeln('CHKCIRC',patch,'   ',SW1,'   ',SW2);

   NE1:= PNTR[ N, PNTR[ E, patch]];
   NE2:= PNTR[ E, PNTR[ N, patch]];
   if NE1 <> NE2 then writeln('CHKCIRC',patch,'   ',NE1,'   ',NE2);

   SE1:= PNTR[ S, PNTR[ E, patch]];
   SE2:= PNTR[ E, PNTR[ S, patch]];
   if SE1 <> SE2 then writeln('CHKCIRC',patch,'   ',SE1,'   ',SE2);
end;

procedure CHKORDER (ord :order);
    { --This procedure checks the consistency
      -- of the order array.
    }
begin
   if (ord[1] + ord[3] = 5) then else
   if (ord[2] + ord[4] = 5) then else
   if (ord[1]=1) or (ord[3]=1) then else
   if (ord[2]=2) or (ord[4]=2) then else
   begin
      writeln(ord[1],' ',ord[2],' ',ord[3],' ',ord[4]);
      error(nihil,'wrong order','CHKORDER');
   end
end ;



var
   k,pp :integer;

begin { -- example main program -- }

   InizData;
   GetCylinder(5,5);

   writeln('level 0');
   for k:= 1 to NumberOfPatches do Show(k);

   writeln('MAKE AN IRREGULAR FIGURE');
   KillCell(15);
   KillCell(21);
   KillCell(24);
   KillCell(27);
   KillCell(28);
   KillCell(29);
   KillCell(31);
   KillCell(39);
   KillCell(47);
   for k:= 1 to NumberOfPatches do Show(k);
```

```
   writeln('Offspring, level 1');
   Scan(RootPointer,NormalOrder,0,0,Offspring);
   for k:= 1 to NumberOfPatches do Show(k);


   writeln('KILL OFFSPRING');
   for k:= NE to NW do PPTY[Sentenced,PNTR[k,17]]:= true;
   RemoveOffspring(17);
   for k:= NE to NW do PPTY[Sentenced,PNTR[k,18]]:= true;
   RemoveOffspring(18);
   for k:= 1 to NumberOfPatches do Show(k);

   Offspring(119);
   Offspring(103);
   Offspring(101);
   Offspring(106);
   Offspring(111);
   Offspring(105);
   Offspring(107);
   Offspring(110);
   Offspring(109);
   Offspring(017);

   ReportIt(0,'cel.cp0');
   ReportIt(1,'cel.cp1');
   ReportIt(2,'cel.cp2');
end.
```

In the figures 2-5, that follow on page 34, the data structure is shown that is generated as an example by the above main program. The numbers in the figures are patch numbers. Solid lines denote boundary walls, dotted lines internal walls. A green wall is identified by an arrow. Green cells are denoted by a G; boundary cells by a B; complete patches contain a C. Notice that boundary cells and green cells are always complete. (In the figures the B overprint the C.) Further, boundary points and green points are identified by B or G respectively.

4 The FORTRAN program

The routines that are to be used to handle the data structure are collected in the file 'basis.f'. The description of the important routines in this file is given in the following comment lines in Section 4.1. To use these routines it is also important to know the names of the global variables. These global variables are found in the include file 'basis.i'. This include file is to be included in each routine that makes use of the data structure. The text of the include file is given in Section 4.2.

4.1 The FORTRAN routines and variables

The following text describes the routines and variables available from the FORTRAN implementation. The text is part of the comment text in the actual program.

```
c     The datastructure is kept in the arrays:
c
c     integer          PNTR (FstPtr:LstPtr, 0:MNOP)  the pointers
c                                                     to the patches
c     logical          PPTY (FstPpt:LstPpt, 0:MNOP)  the properties
c     double precision DATA (1:MNOD, 0:MNOP)         the data
c
c
c
c   The data are handled by the following subroutines:
c
c
c   A SUBROUTINE TO INITIALIZSE THE DATA STRUCTURE
c
c     subroutine DatIni(MNOPac, MNODac, PNTR, PPTY, DATA)
c   A routine to be called once during a run, before the
c   data structure is used.
c   MNOPac and MNODac define the actual MNOP and MNOD used.
c   MNOP: Maximum Number Of Patches.
c   MNOD: Maximum Number Of real Data per patch.
c   (The arrays PNTR, PPTY and DATA should have at least the
c   size specified above.)
c
c
c   SUBROUTINES FOR THE CONSTRUCTION OF A DOMAIN
c
c     subroutine MkRec(m, n, PNTR, PPTY, DATA)
c   This subroutine constructs a data structure corresponding to a
c   rectangular domain, with (on level 0)
c   m cells in the Xi-direction and
c   n cells in the Eta-direction.
c   For the relation between (Xi,Eta)-coordinates, cell numbering
c   and physical coordinates we refer to [1].
c
c
c     subroutine MkCyl(m, n, PNTR, PPTY, DATA)
c   This subroutine constructs a data structure corresponding to a
c   cylindrical domain, with (on level 0)
c   m cells in the Xi-direction and
c   n cells in the Eta-direction.
c   The cylindrical domain is constructed such that the patches at
```

```
c   maximum Eta are the bottom neighbours of the patches at Eta equal
c   zero.
c
c     subroutine RmCel0(i, j, PNTR, PPTY, DATA)
c   A subroutine to re-shape the domain on level 0. This
c   subroutine is to be called for each cell that is to be removed
c   from the rectangle or cylinder that was made by MkRec or MkCyl.
c   The cell is denoted by (i,j), the (Xi,Eta)-coordinates of
c   the complete patch containing the cell.
c
c
c   SUBROUTINES TO ADD TO OR REMOVE FROM THE DATA STRUCTURE
c
c     subroutine MkOfsp(daddy, PNTR, PPTY, DATA)
c   A subroutine that creates four new cells as kids of the cell
c   identified by the pointer 'daddy'.
c
c     subroutine RmOfsp(daddy, PNTR, PPTY, DATA)
c   A subroutine that removes the kid-cells of the cell identified by
c   the pointer 'daddy', provided that all these kid-cells are marked
c   as 'Sentenced'.
c
c
c   A SUBROUTINE TO SCAN THE PATCHES IN THE DATA STRUCTURE
c
c     subroutine Scan(RtPt, order, FrmLv, ToLv, DoIt, PNTR, PPTY, DATA)
c   A subroutine that scans all patches that are decendents of the
c   patch denoted by the pointer 'RtPt', and that are on a level 'lv'
c   for which 0  <=  FrmLv  <= lv  <=  ToLv .
c   The patches are visited tree-wise, in a sequence determined by
c   'order'.
c   At each patch visited a call is made to the subroutine 'DoIt':
c     call DoIt(patch, PNTR, PPTY, DATA)
c   where 'patch' identifies the patch visited.
c   The subroutine 'DoIt' can be any subroutine constructed by the
c   user, provided that it has the above syntaxis.
c   If necessary, additional communication between the actual subroutine
c   'DoIt' and the (sub)program calling 'Scan' can be taken care of
c   by a locally defined COMMON block, only known by the (sub)program
c   calling 'Scan' and the actual 'DoIt'.
c
c
c   SOME ADDITIONAL SUBROUTINES
c
c     subroutine Error(integer, string1, string2)
c   A Subroutine called after a fatal error in the program occurred.
c   The integer and strings are printed on standard output.
c
c     subroutine Warnin(integer, string1, string2)
c   A subroutine called after a non-fatal error in the program occurred.
c   The integer and strings are printed on standard output.
c
c     subroutine Show(patch, PNTR, PPTY, DATA)
c   A subroutine to print the PNTR and PPTY data of the patch 'patch'.
c   These data are given as one line of output on the standard output.
```

```
c
c
c   COMMON BLOCK FOR THE DATA STRUCTURE
c
c     common /DatGlb/ MNOP, MNOD,
c    +                RtLv, LstSpa, NOP, SizeX0, SizeY0, NrmOrd
c   The passing of global properties of the data structure to
c   subroutines is provided by the COMMON block 'DatGlb'.
c
c
c   Meaning of some global parameters and variables:
c
c
c   VARIABLES IN THE COMMON BLOCK 'DatGlb'
c
c   MNOP        Maximum Number Of Patches
c   MNOD        Number Of Data (of a patch)
c   RtLv        Root Level
c   LstSpa      Last Space
c   NOP         Number Of Patches
c   SizeX0      rectangle Size Xi-direction on level 0
c   SizeY0      rectangle Size Eta-direction on level 0
c
c   POINTER PARAMETERS
c   Nil         Nil Pointer
c   RtPtr       Root Pointer
c
c   POINTER ARRAY BOUND PARAMETERS
c   FstPtr      First Pointer
c   LstPtr      Last Pointer
c
c   POINTER ARRAY INDEX PARAMETERS
c   LV          Level
c   XX          Xi-coordinate
c   YY          Eta-coordinate
c   PT          Parent
c   NE          North-East (kid)
c   SE          South-East (kid)
c   SW          South-West (kid)
c   NW          North-West (kid)
c   NN          North
c   EE          East
c   SS          South
c   WW          West
c
c   PROPERTY ARRAY BOUND PARAMETERS
c   FstPpt      First Property
c   LstPpt      Last Property
c
c   PROPERTY ARRAY INDEX PARAMETERS
c   Compl       Complete
c   WallH       Horizontal Wall
c   WallV       Vertical Wall
c   BdyPnt      Boundary Point
c   BdyWaH      Boundary Wall Horizontal
```

```
c   BdyWaV      Boundary Wall Vertical
c   BdyCel      Boundary Cell
c   GrnPnt      Green Point
c   GrnWaH      Green Wall Horizontal
c   GrnWaV      Green Wall Vertical
c   GrnCel      Green Cell
c   Prgnt       Pregnant
c   Sntncd      Sentenced
c   Dead        Dead
c
c   A SCANNING ORDER
c
c   NrmOrd    Normal Order ( = SW, NW, SE, NE )
c   This order used by 'Scan' causes the patches to be visited
c   from the SW- to the NE-corner of the domain. In some subroutines
c   using 'Scan', this scanning order is essential and therefore should
c   not be changed.
c
c   INCLUDE FILE
c
c   In order to create a compact source-code file, the subroutines
c   make use of an 'include file'. The way of including this file
c   by a compiler directive such as 'include', depends on the machine
c   and FORTRAN compiler being used. The FORTRAN-code becomes ANSI-
c   compatible by once and for all copying the include file into
c   the subroutines, at the location where it is now included.
```

4.2 The FORTRAN include file

The following text is the contents of the include file "basis.i", that is used to have common identifiers, with the same meaning, for all soubroutines that make use of the data structure.

```
 integer          FstPtr, LstPtr,
+                 LV, XX, YY, PT,
+                 NE, SE, SW, NW, NN, EE, SS, WW,
+                 FstPpt, LstPpt,
+                 Compl, WallH, WallV,
+                 BdyPnt, BdyWaH, BdyWaV, BdyCel,
+                 GrnPnt, GrnWaH, GrnWaV, GrnCel,
+                 Prgnt, Sntncd, Dead,
+                 Nil, RtPtr
 parameter        (FstPtr =-3, LstPtr = 8,
+                 LV     =-3, XX     =-2, YY     =-1,  PT     = 0,
+                 NE     = 1, SE     = 2, SW     = 3,  NW     = 4,
+                 NN     = 5, EE     = 6, SS     = 7,  WW     = 8,
+                 FstPpt = 1, LstPpt =14,
+                 Compl  = 1, WallH  = 2, WallV  = 3,
+                 BdyPnt = 4, BdyWaH = 5, BdyWaV = 6,  BdyCel = 7,
+                 GrnPnt = 8, GrnWaH = 9, GrnWaV = 10, GrnCel = 11,
+                 Prgnt  =12, Sntncd =13, Dead   = 14,
+                 Nil    = 0, RtPtr  = 1)

 integer          MNOP, MNOD,
+                 RtLv, LstSpa, NOP, SizeX0, SizeY0, NrmOrd(4)
 common /DatGlb/  MNOP, MNOD,
+                 RtLv, LstSpa, NOP, SizeX0, SizeY0, NrmOrd

 integer          PNTR(FstPtr:LstPtr, 0:MNOP)
 logical          PPTY(FstPpt:LstPpt, 0:MNOP)
 double precision DATA(1:MNOD, 0:MNOP)
```

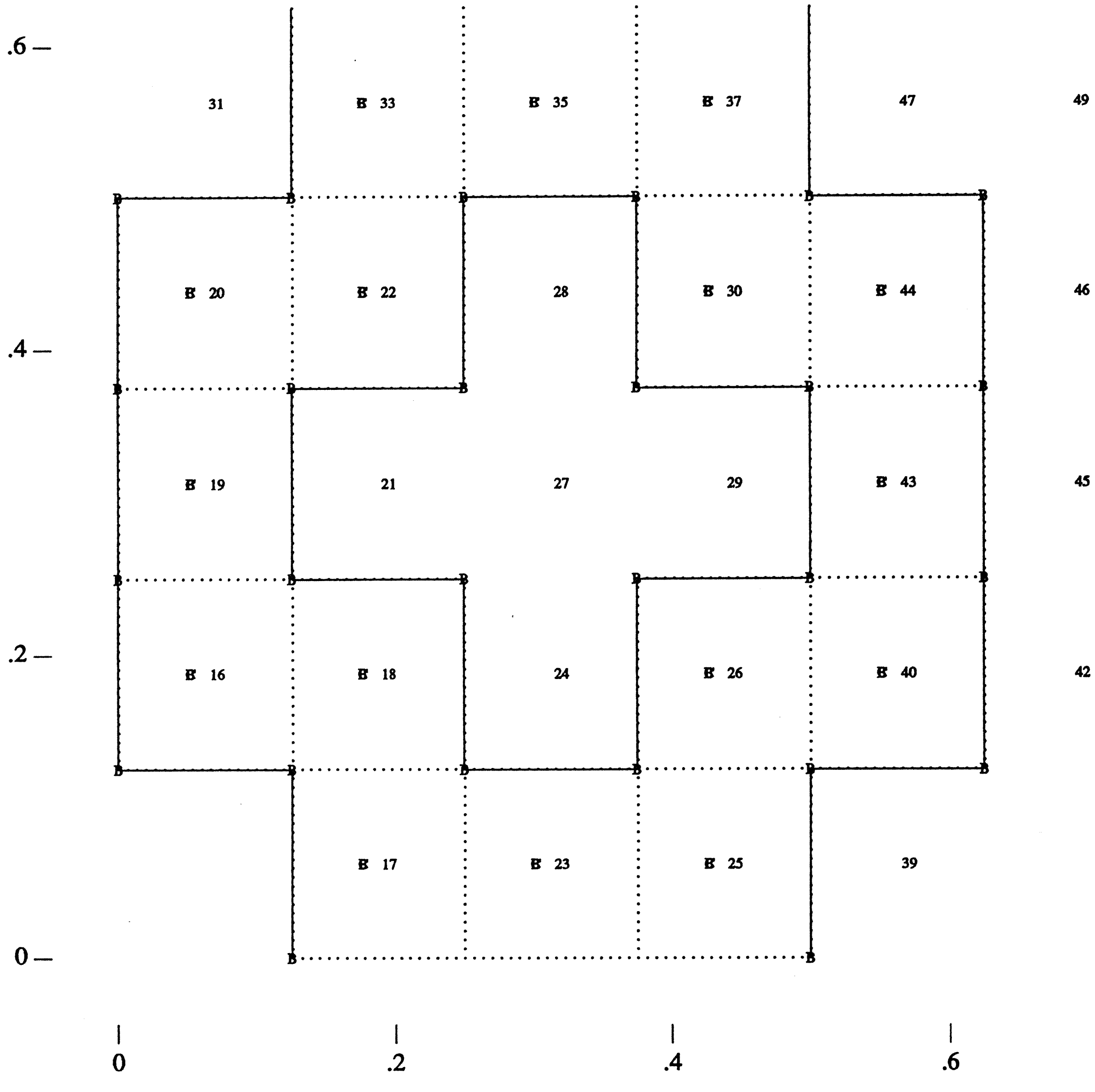

Figure 2: The example (see page 28): all cells on level 0. See section 3.5.

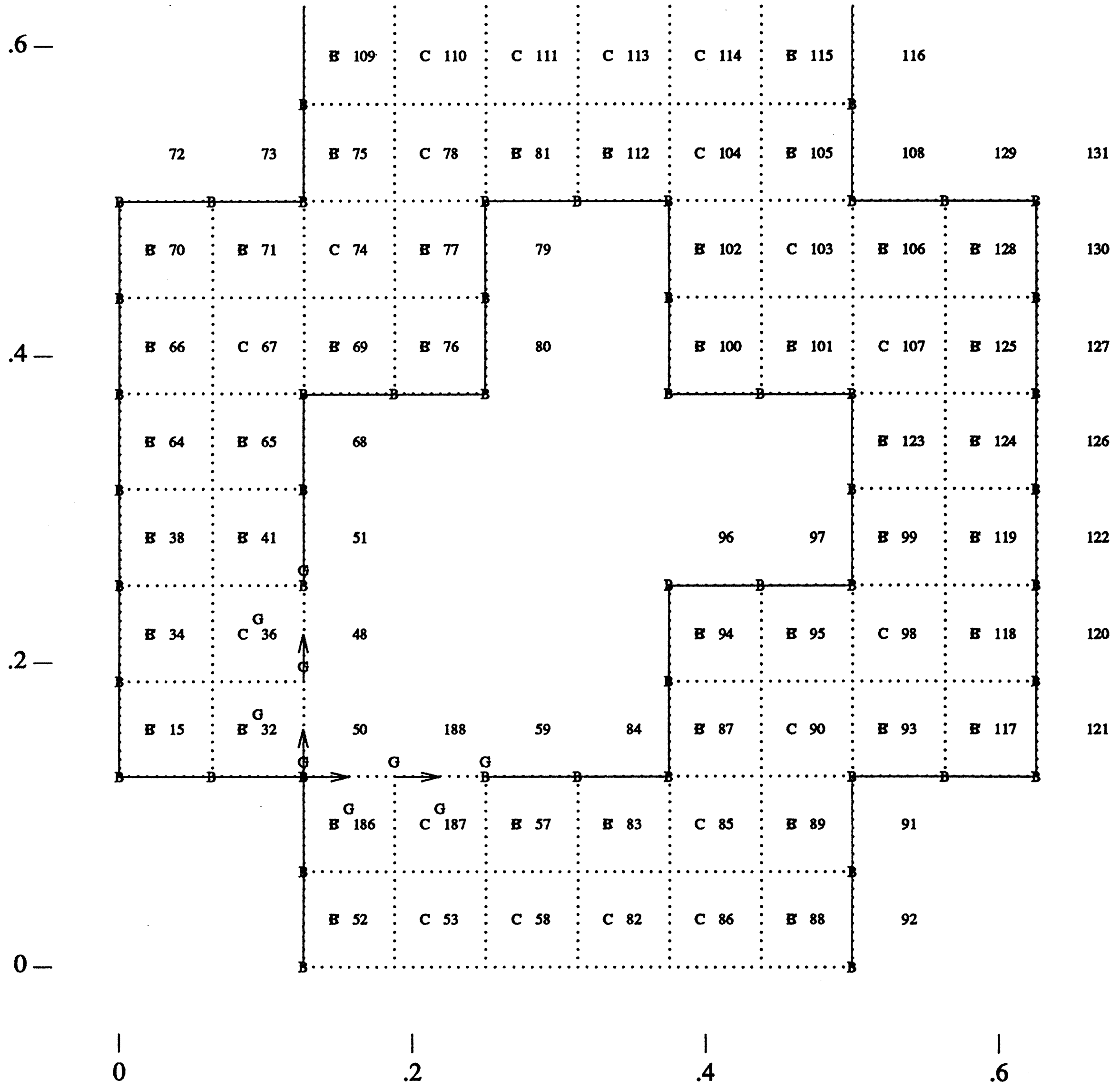

Figure 3: The example: all cells on level 1. See section 3.5.

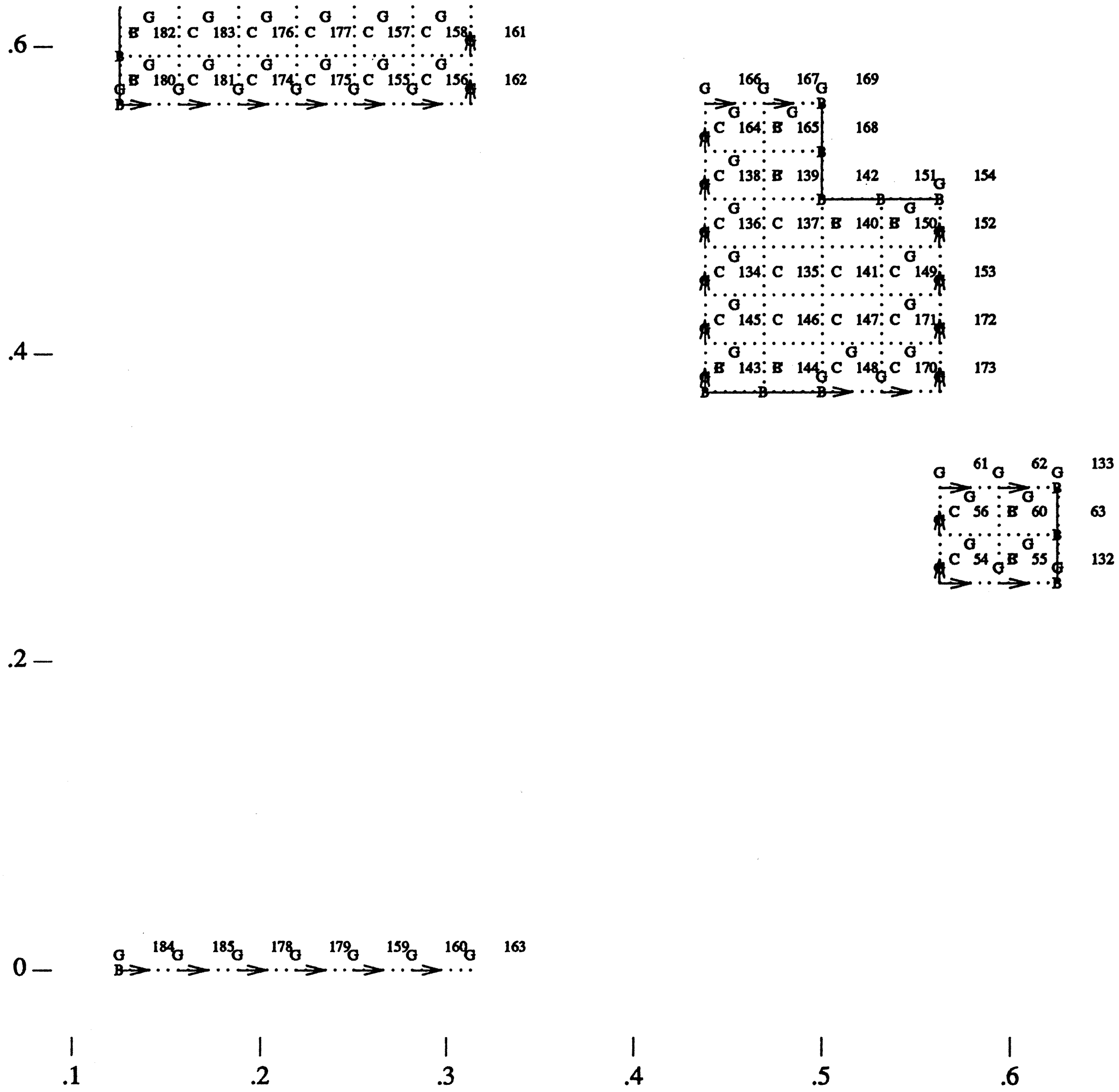

Figure 4: The example: all cells on level 2. See section 3.5.

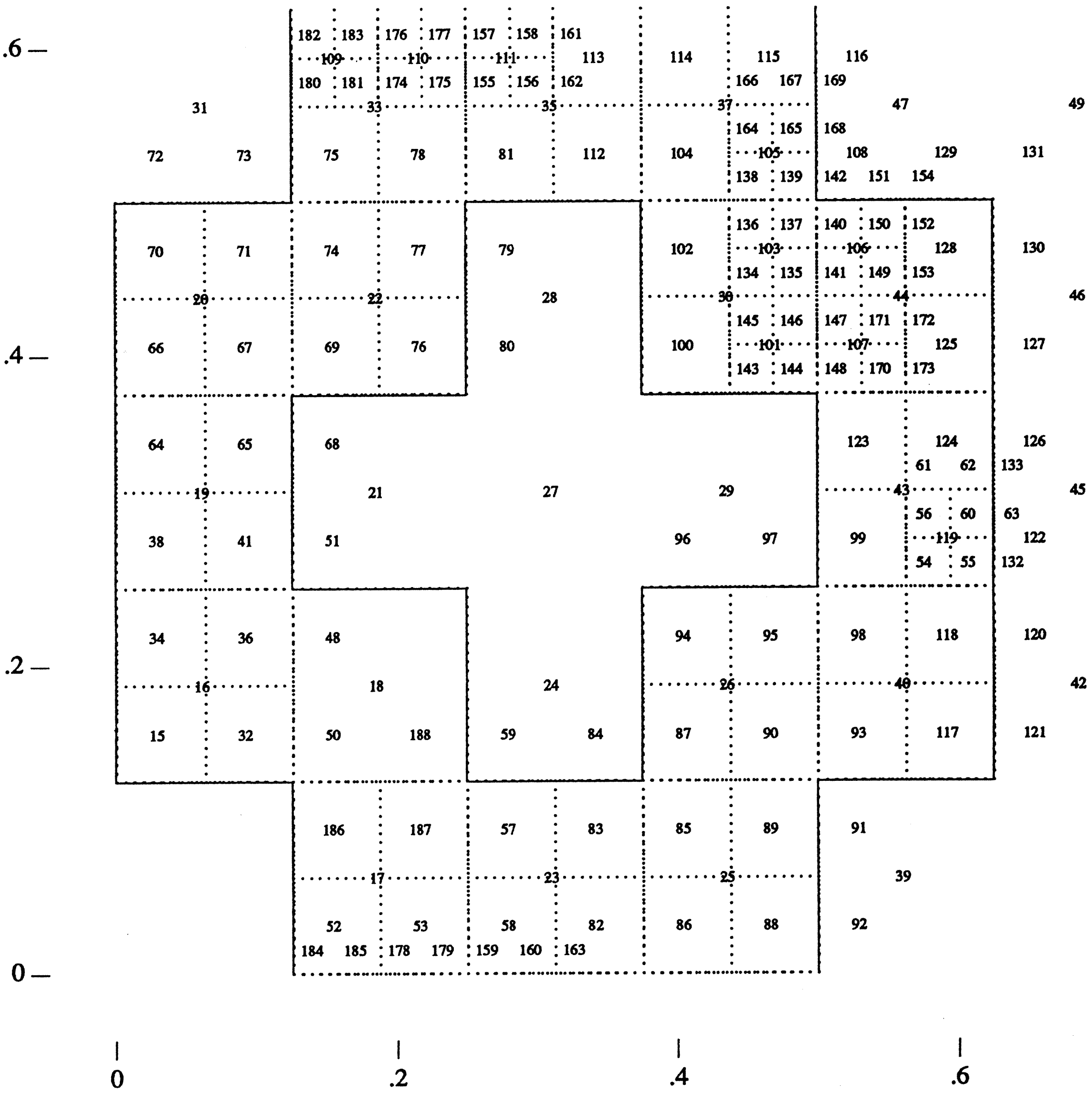

Figure 5: The example: all cells. See section 3.5.

Index

Contents