

## AN ALGORITHM FOR MESH REFINEMENT AND UN-REFINEMENT IN FAST TRANSIENT DYNAMICS

FOLCO CASADEI

*Joint Research Center, Institute for the Protection and Security of the Citizen,  
T.P. 480, I-21027 Ispra, Italy  
folco.casadei@jrc.ec.europa.eu  
http://europlexus.jrc.ec.europa.eu/*

PEDRO DíEZ, FRANCESC VERDUGO

*Universitat Politècnica de Catalunya,  
Barcelona, Spain  
vre.diez@upc.edu, francesc.verdugo@upc.edu*

Received (Day Month Year)

Revised (Day Month Year)

A procedure to locally refine and un-refine an unstructured computational grid of four-node quadrilaterals (in 2D) or of eight-node hexahedra (in 3D) is presented. The chosen refinement strategy generates only elements of the same type as their parents, but also produces so-called hanging nodes along non-conforming element-to-element interfaces. Continuity of the solution across such interfaces is enforced strongly by Lagrange multipliers. The element split and un-split algorithm is entirely integer-based. It relies only upon element connectivity and makes no use of nodal coordinates or other real-number quantities. The chosen data structure and the continuous tracking of the nature of each node facilitate the treatment of natural and essential boundary conditions in adaptivity. A generalization of the concept of neighbor elements allows transport calculations in adaptive fluid calculations. The proposed procedure is tested in structure and fluid wave propagation problems in explicit transient dynamics.

*Keywords:* Mesh refinement, mesh un-refinement, adaptivity, 3D, explicit, transient dynamics.

### 1. Introduction

The numerical simulation of complex 3D fast transient dynamic phenomena, e.g. for the prediction of blast effects on critical infrastructures, requires long calculations even on today's computers, due to the large number of elements—typically in the order of millions or even more—needed to obtain the desired accuracy. One of the most promising techniques to save CPU time is mesh adaptivity, i.e. automatic mesh refinement and un-refinement in order to “put the small elements only where they are really needed”. Adaptive techniques based on error estimators/indicators are nowadays relatively common in statics, but their application in fast transient dynamic problems, characterized by wave propagation, is still challenging. In order to follow rapidly

evolving phenomena, the chosen mesh adaptation techniques must be particularly simple, efficient and robust.

This paper presents a strategy to continuously refine and un-refine a computational mesh in explicit fast transient dynamics. The element shapes considered here are only the 4-node quadrilateral (QUA4) in 2D and the 8-node hexahedron (CUB8) in 3D. However, the method can be applied with minor modifications also to other element shapes. One basic choice is that, when splitting an element in order to locally refine the mesh, only elements of the same shape as their “parent” (i.e. either QUA4 or CUB8) are generated. This simplifies the geometry- and connectivity-updating calculations and contributes to the robustness of the method, but also produces so-called “hanging” nodes along the non-conforming element-to-element interfaces, i.e. in the zones where element size varies.

Over the last three decades a vast literature has been produced on mesh refinement techniques related to adaptivity, and a review is out of scope here. As concerns the purely geometric aspects of element splitting and un-splitting and the tree-like data organization, the strategy chosen here resembles the one proposed for example in 2D by Yerry and Shephard already in 1983 and by Demkowicz, Oden *et al.* [1985; 1989]. Similar techniques are still used nowadays in complex 3D applications; see *e.g.* Meyer [2009] or Burstedde *et al.* [2009], where massive parallelization aspects are discussed.

Our final target is to use mesh adaptivity in safety studies to evaluate, in particular, the vulnerability of buildings or other critical infrastructure to blast loading. These applications require modeling of the fluid and of the structure (including failure and fragmentation), plus robust and efficient fluid-structure interaction (FSI) algorithms. Numerical models are huge in 3D, with millions of finite elements or finite volume cells, mainly in the fluid domain. Therefore, adaptivity can be exploited in a variety of manners. For example: to accurately track the shock fronts of blast waves, to improve the representation of material interfaces and of free surfaces (*e.g.* in blast loading of submerged structures), and to automatically refine the fluid mesh near structural walls for improved accuracy of embedded-type FSI algorithms. The first task requires suitable error indicators, see *e.g.* the pioneering work of Peraire *et al.* [1987], the comprehensive paper by Nithiarasu and Zienkiewicz [2000] and the article of Frey and Alauzet [2005], for fluid problems, or the recent paper by Erhart, Wall and Ramm [2006] on large deformation under impact, for solid problems.

The present work deals with only one ingredient of adaptive formulations, i.e. the geometrical and data-structure aspects related to continuous mesh refinement and un-refinement during a transient dynamic solution. The subject of error estimators/indicators—or of any other criteria (*e.g.* structure proximity in FSI) needed to automatically drive mesh adaptation—is left for a subsequent contribution.

The class of problems of interest here, namely fast transient FSI phenomena related to blast loading of complex 3D structures up to failure and fragmentation, can partially justify the specific choice of elements shape (quadrilaterals/hexahedra instead of the simpler and more commonly used, in adaptivity, triangles/tetrahedra) and of refinement strategy (which produces non-conforming refined meshes due to hanging nodes). In this class of applications use is typically made of “embedded” FSI algorithms, see *e.g.*

Casadei *et al.* [2011], whereby the structure mesh is immersed in a regular (even uniform sometimes) background fluid mesh. This explains the use of quadrilaterals/hexahedra, without need for complicated mesh generation tools that typically operate only on simplexes (triangles/tetrahedra). Although structure adaptivity is also considered below for full generality, we are thus mainly interested in adapting the fluid domain.

The interested reader may find it useful to compare the present mesh refinement strategy with the more widely used ones, based upon simplex elements, e.g. in the papers by Liu, Zhang and co-workers. The use of simplex shapes facilitates automatic meshing and re-meshing of arbitrarily complex domains, whenever this is needed: not only for the discretization of the computational domain in FE/FV solid mechanics [Zhang, Liu *et al.* (2008-2011), Nguyen-Thoi, Liu *et al.* (2009)] and in fluids [Xu, Liu *et al.* (2010)], but also in those mesh-free formulations which still require a background mesh to perform numerical integration [Liu and Tu (2002), Liu, Kee *et al.* (2006, 2008)].

Mesh refinement in the cited references makes typically use of Delaunay triangulation and/or of its dual, the Voronoi diagram, in order to obtain optimal and conforming refined (arbitrary, unstructured) meshes. This may sometimes lead to badly-shaped (highly distorted) elements, thus requiring additional treatments such as redundant cells removal, diagonal swapping or grid smoothing. In the present approach refinement is conceptually straightforward, being based on simple bisection. This generates only descendent elements of the same shape and aspect as their parent (which is important, especially in 3D) and avoids any element distortion by construction. However, the price to be paid is the appearance of hanging nodes (non-conforming interfaces), which require a specific treatment, e.g. by Lagrange multipliers as proposed below. It is also clear that such a technique is especially useful when the base mesh is regular, although not necessarily structured (which is the case in the applications envisaged here). With geometrically complex domains to be discretized by conforming meshes, the use of simplexes is obviously superior. As concerns efficiency of the proposed procedure, no comparison with other methods (e.g. based on simplexes) was attempted. Despite its simplicity (bisection) the present procedure is relatively involved in 3D, due to hanging nodes. However, the resulting information on the nature of nodes and of neighboring elements greatly facilitates the treatment of boundary conditions (and of transport terms in fluids), thus recovering part of the effort devoted to geometrical calculations.

This paper is organized as follows. Section 2 presents the chosen mesh refinement strategy, the classification of nodes—in such a way to facilitate the treatment of boundary conditions and the enforcement of continuity constraints—and the neighborhood relations between elements, which are useful both for element adaptation and for the calculation of numerical fluxes in fluid problems. Then in Section 3 the element splitting and un-splitting algorithms are detailed. Section 4 shows how to impose constraints on the adaptive solution according to the nature of each node resulting from the algorithms of Section 3. The calculation of numerical fluxes in adaptive fluid meshes using a generalized notion of neighbors is also detailed. Numerical examples are presented in Section 5 and conclusions and perspectives for future developments are given in Section 6. Some auxiliary procedures are listed in the Appendix.

The proposed algorithms are implemented and tested in EUROPLEXUS [Casadei *et al.* (2012)], a computer program for fast transient analysis of fluid-structure systems under dynamic loading, which is jointly developed by the French Commissariat à l’Energie Atomique (CEA Saclay) and by the Joint Research Centre of the European Commission (JRC Ispra).

## 2. Mesh refinement strategy

The chosen mesh refinement strategy is illustrated in 2D for the QUA4 element (Fig. 1).

### 2.1. Base mesh

First, some terminology is introduced for the *base* mesh, i.e. the initial (coarse) mesh, assumed to be given in input (left drawing in Fig. 1). Let  $i$  be the generic *element*, of *vertices*  $V_I$ ,  $I = 1, K, 4$  which coincide with the four *element nodes*  $I, J, K, L$ . Elements are numbered anti-clockwise, e.g.  $IJKL$  for element  $i$  in the figure. Each element has four *faces*  $F_k$ ,  $k = 1, K, 4$ , with two nodes each:  $IJ, JK, KL, LI$  in the example. The mesh information is completed by the list of neighbor elements, or simply *neighbors*, across each face. This list is built up once by performing a search over the base mesh, after reading it as input data. Two elements are (reciprocally) neighbors across a given face if the face belongs to both elements (with opposite orientations). In Fig. 1 element  $i$  has no neighbor across faces 1 and 2 while it has neighbors  $j$  and  $l$  across faces 3 and 4, respectively. The presence of a neighbor indicates an *internal* face (within the domain), while its absence indicates an *external* face (on the domain boundary). The base mesh is assumed *conforming*, hence neighbors are reciprocal: if  $i$  has neighbor  $j$  across one of its faces  $F_k^{(i)}$ , then  $j$  has neighbor  $i$  across one of its faces  $F_l^{(j)}$ , and *vice versa*. Note also that the base mesh is assumed *unstructured* for full generality, although all numerical examples in Section 5 use structured element patches for simplicity.

### 2.2. Element splitting

A generic element  $i$  is split as shown in Fig. 1, right drawing. Four *descendent* elements  $i_1, i_2, i_3, i_4$  are created, all of which have element  $i$  as their parent element. In addition,

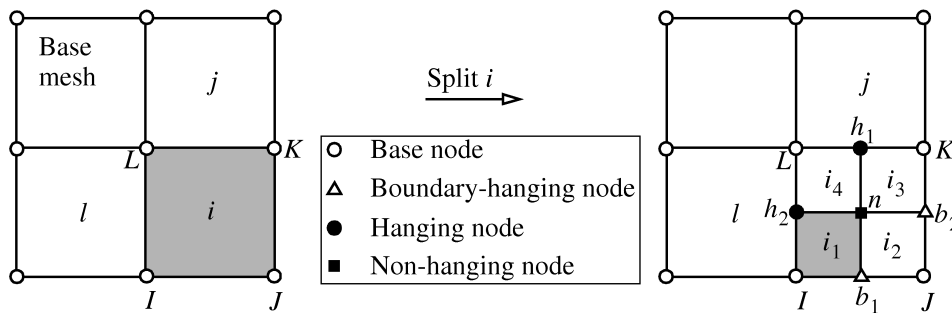


Fig. 1. Splitting a QUA4.

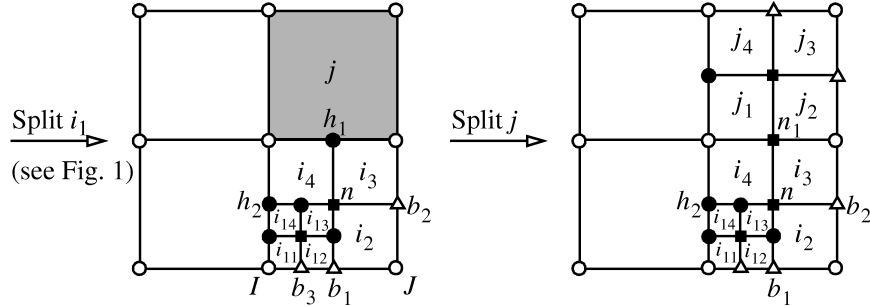


Fig. 2. Further splitting.

one to five new descendent nodes are created. In the example,  $n$  is the *central node*, which is always created, while  $b_1, b_2, h_1, h_2$  are the *face nodes*. A new face node is created or not, depending upon the mesh state across the face itself. If there is a neighbor, and if this has been previously refined, then the face node exists already; otherwise, a new face node is created. In the example of Fig. 1 all four face nodes are created upon splitting of element  $i$ .

Element splitting is a recursive process, and can go on as shown e.g. in Fig. 2. Therefore, it seems convenient to store elements in a tree-like data structure [Yerry and Shephard (1983)], see Fig. 3 corresponding to the last mesh of Fig. 2. Each element occupies a certain *level* in the tree, with base elements (and only them) at level 1 by convention. Base elements have no parent. The notions of *right* and *left siblings* are also useful, to traverse the tree quickly in both directions. Elements with descendants are called *branches* while elements without descendants are called *leaves*. Note that only leaf elements take part in the computation. Branch elements are not computed, but are kept in memory (flagged as idle). This facilitates mesh un-refinement by simply re-activating a previously idle element.

In Fig. 2 the example of Fig. 1 is continued by showing further splitting of level-2 element  $i_1$ , which generates level-3 descendants  $i_{11}, i_{12}, i_{13}, i_{14}$  and five new nodes. Then, base element  $j$  is split into  $j_1, j_2, j_3, j_4$ , showing a case where one of the face nodes ( $h_1$ ) exists already and needs not be created. However, the nature of this node *changes* from hanging to non-hanging (see Section 2.3) and therefore it is renamed  $n_1$  for clarity.

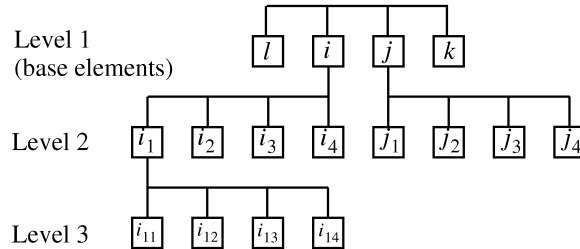
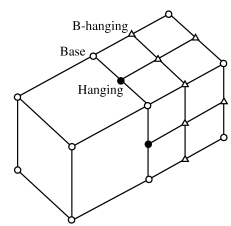


Fig. 3. Tree data structure for the elements.

### 2.3. Node types

With the chosen mesh refinement strategy there are up to *four* different types of nodes in a mesh, and the nature of each node is (continuously) tracked during the transient. Like for elements, a level is associated to each node. The nodes of the base mesh are called *base nodes* and are the only nodes at level 1. All other nodes are *descendent nodes*, of which there are three types: *hanging* nodes, *boundary-hanging* nodes and *non-hanging* nodes, see Table 1.

Table 1. A classification of nodes.

Node type	Location	Classification	Interface	 <p>Example of hanging nodes on 3D boundary</p>
Base	Internal	—	—	
	Boundary	—	—	
Descendent	Internal	Non-hanging	Conforming	
		Hanging	Non-conforming	
	Boundary	Hanging (3D only)	Non-conforming (internally)	
		B-hanging	—	

Hanging nodes, e.g.  $h_1$  and  $h_2$  in Fig. 1, are a direct consequence of the chosen element splitting procedure. They occur at locally non-conforming element-to-element interfaces, i.e. wherever a “bigger” and a “smaller” element are contiguous but without sharing a common face. Suitable constraints have to be imposed at such nodes in order to ensure continuity of the numerical solution across the interface. For example, in a displacement-based FE formulation it is clear that the degrees of freedom of  $h_1$  in Fig. 1 are not free. They depend (“hang”) upon those of nodes  $L$  and  $K$  (i.e. the nodes of the face on which the hanging node is located), if continuity has to be ensured. These nodes are called the *masters* of the hanging node.

Boundary-hanging nodes (*b-hanging* for brevity in the following), e.g.  $b_1$  and  $b_2$  in Fig. 1, are the descendent nodes, located on the boundary of the body (as the name implies), which lie on a conforming element-to-element interface. Note that in 2D a descendent node on the boundary is guaranteed to lie upon a conforming interface. Consequently, in 2D all descendent nodes on the boundary are b-hanging. However, in 3D the same property does *not* hold (see the inset in Table 1 and the discussion in Section 3), and therefore *both* conditions must be satisfied for a 3D node to be b-hanging. B-hanging nodes do not really “hang” upon any other nodes (and therefore the term is perhaps somewhat misleading), but their identification greatly facilitates the treatment of boundary conditions in an adaptive mesh. These nodes can be programmed so as to automatically “inherit” any boundary conditions that a user may have prescribed on the base mesh, i.e. on the nodes of the *base face* upon which the b-hanging node is located. Such nodes are called the *masters* of the b-hanging node. For example, in Fig. 2 nodes  $b_1$  and  $b_3$  inherit conditions from base nodes  $I$  and  $J$ . Note that the masters of a b-hanging node are always base (boundary) nodes, while the masters of a hanging node can

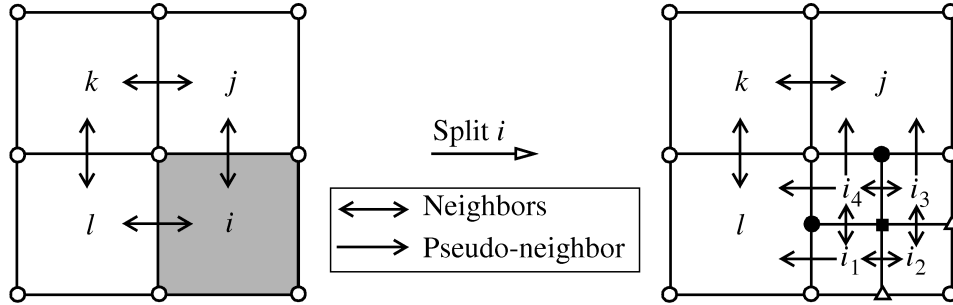


Fig. 4. Neighbors and pseudo-neighbors.

be of any type. For the element shapes considered here, each hanging or b-hanging node has two masters in 2D, two or four masters (depending on whether it is on a corner or on a face) in 3D.

Non-hanging nodes are any other descendent nodes. They are necessarily located in the interior of the domain, at locally conforming element-to-element interfaces. The degrees of freedom associated with these nodes are completely free. In fact, the case of “boundary” conditions (constraints) imposed by a user on *internal* nodes is not considered in this paper.

#### 2.4. Neighbors and pseudo-neighbors

Another consequence of the chosen element splitting procedure is that the simple bi-univocal neighborhood relations valid for the (conforming) base mesh no longer hold for a locally refined mesh, see Fig. 4. Neighbors are connected by a double arrow as in  $i \leftrightarrow j$ . When element  $i$  is split into its descendents, there are *two* of these (smaller) elements  $i_3$  and  $i_4$  adjacent to element  $j$ . We identify this situation by saying that  $i_3$  and  $i_4$  have  $j$  as *pseudo-neighbor* (or simply *p-neighbor* for brevity), across their upper face in the example. Note that this relation is univocal: element  $j$  does *not* have  $i_3$  and  $i_4$  as p-neighbors across its bottom face in the example. In fact, it (still) has element  $i$  (their parent, now idle) as neighbor. P-neighbors are connected by a single arrow as in  $i_4 \rightarrow j$ . The following definitions of neighbor and of p-neighbor are adopted:

- The *neighbor* of an element across a face is the element of the same level, active (leaf) or idle (branch), adjacent to the face, or 0 if there is no such element.
- The *p-neighbor* of an element across a given face is the lower-level (i.e. larger) active (leaf) element adjacent to the face, or 0 if there is no such element.

Neighbor and p-neighbor are mutually exclusive at a given face. That is, an element can have either a neighbor, or a p-neighbor (or nothing) across a face, but it cannot have both. This choice simplifies the implementation and speeds up calculations because the table of neighbors and p-neighbors has a fixed, known length. The presence of a neighbor across a face indicates an internal, locally conforming element-to-element interface. The presence of a p-neighbor indicates an internal, locally non-conforming interface. In fact, on the concerned face (of the p-neighbor) there are always one or more hanging nodes;

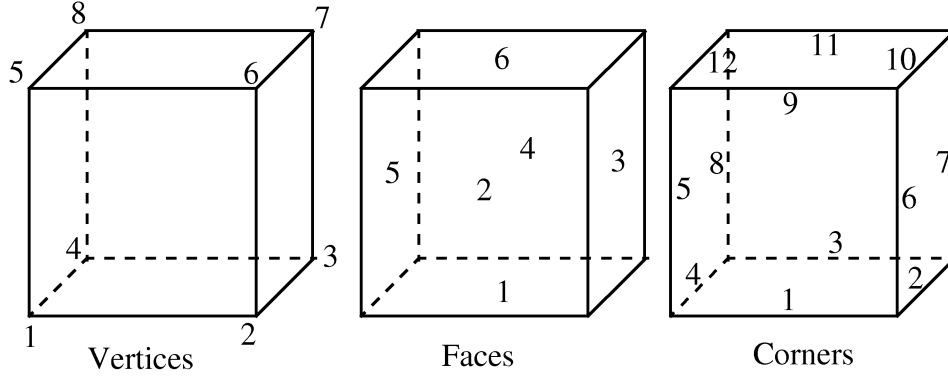


Fig. 5. CUB8 vertices, faces, corners.

see the examples. The absence of a neighbor and p-neighbor indicates that the face is external. The p-neighbor relation is univocal according to the above definition. That is, if element  $i$  has  $j$  as p-neighbor, then  $j$  has  $i$  neither as p-neighbor, nor as neighbor. In fact, in this case  $j$  has an ancestor of  $i$  (more precisely, the one at the same level as  $j$ ) as neighbor.

Keeping track of neighbors and p-neighbors during mesh refinement and un-refinement serves two purposes: first, it is used in the mesh adapting algorithms themselves; second, it allows efficient computation of transport terms in fluid applications, see Section 4.3.

### 3. Element splitting and un-splitting algorithms

The element splitting and un-splitting algorithms are now presented, for generality in 3D for the CUB8 element shape, i.e. the 8-node hexahedron. The same algorithms are applied also in 2D to the QUA4 element shape, with the simplifications indicated in Section 3.7. Extensions to other 2D or 3D element shapes are also possible using the same strategy, with only minor modifications. Before detailing the algorithms, some further definitions are given and the data structure used in the implementation is shortly introduced.

#### 3.1. The CUB8 hexahedron

The CUB8 element is shown in Fig. 5. It has eight vertices (or nodes)  $V_I$ ,  $I = 1, K, 8$  enumerated (in the element connectivity table) in such a way that the first four are located on the “bottom” element face, and so that the oriented normal to this face “enters” into the element. The last four vertices, located on the “upper” element face, are enumerated consistently (in this case the oriented normal “exits” from the element) so that  $V_5$  stays “above”  $V_1$ , etc.

The element has 6 faces  $F_k$ ,  $k = 1, K, 6$ , with four nodes each. Face numbering is such that the oriented normal always exits from the element. This convention is called



*anti-clockwise* orientation. For example,  $F_1 = \{V_1, V_4, V_3, V_2\}$ , or  $F_1 = \{1, 4, 3, 2\}$  for brevity. Faces are shown in Fig. 5 and are listed (together with all other constant connectivity data) in Table 2.

Table 2. Constant connectivity data used for the CUB8 element shape.

Data	Symbol	Values
Vertices	$V_{1..8}$	1-2-3-4-5-6-7-8.
Faces	$F_{1..4, 1..6}$	1-4-3-2; 1-2-6-5; 2-3-7-6; 3-4-8-7; 4-1-5-8; 5-6-7-8.
Corners	$C_{1..2, 1..12}$	1-2; 2-3; 3-4; 4-1; 1-5; 2-6; 3-7; 4-8; 5-6; 6-7; 7-8; 8-5.
Faces to corners	$F_C 1..4, 1..6$	4-3-2-1; 1-6-9-5; 2-7-10-6; 3-8-11-7; 4-5-12-8; 9-10-11-12.
Corners to face	$C_F 1..2, 1..12$	1-2; 1-3; 1-4; 1-5; 2-5; 3-2; 4-3; 5-4; 2-6; 3-6; 4-6; 5-6.
Corners to corner nodes	$C_C 1..2, 1..12$	1-2; 2-3; 3-4; 4-1; 1-5; 2-6; 3-7; 4-8; 5-6; 6-7; 7-8; 8-5.
Faces to face nodes	$F_F 1..2, 1..6$	1-3; 1-6; 2-7; 3-8; 4-5; 5-7.
Corners to descendents	$C_{D, 1+2, 1+12}$	1-2; 2-3; 3-4; 4-1; 1-5; 2-6; 3-7; 4-8; 5-6; 6-7; 7-8; 8-5.

3D elements also have *corners*, which do not exist in 2D. In the CUB8 there are twelve corners  $C_i$ ,  $i = 1, K, 12$ , with two nodes each, see Fig. 5. For example,  $C_1 = \{V_1, V_2\} = \{1, 2\}$ . Much of the complexity of 3D algorithms comes from the fact that in order to split or un-split an element one has to consider not only the elements adjacent to faces (i.e. neighbors and p-neighbors), but also “adjacent to corners”, see Section 3.3. This is not the case in 2D.

Each face of the CUB8 is bounded by four corners. This information is kept in a constant table  $F_C(c, f)$ ,  $c = 1, K, 4$ ,  $f = 1, K, 6$  where the first entry is the corner and the second entry is the face. For example, for the first face of the CUB8:  $F_C(c, 1) = \{C_4, C_3, C_2, C_1\} = \{4, 3, 2, 1\}$ . Note that the corners of a given face  $F_k$  are enumerated in the same order as face nodes: i.e. anti-clockwise starting from the first node of the face.

Each corner is adjacent to two faces of the element. This information is kept in a constant table  $C_F(f, c)$ ,  $f = 1, 2$ ,  $c = 1, K, 12$  where the first entry is the face and the second entry is the corner (this is somehow the “inverse” of  $F_C$ ). For example, for the first corner of the CUB8:  $C_F(f, 1) = \{F_1, F_2\} = \{1, 2\}$ .

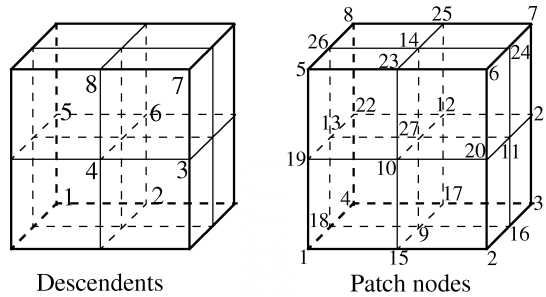


Fig. 6. Splitting/un-splitting a CUB8.

The CUB8 element is split into a *patch* of eight CUB8 *descendents*  $d_j$ ,  $j = 1, K, 8$ , see Fig. 6. They are enumerated in such a way that the  $j$ -th descendent  $d_j$  is adjacent to (contains) the  $j$ -th vertex  $V_j$  of the parent. In the splitting/un-splitting process, the following nodes have to be considered in addition to the cube vertices: one node for each face, called the *face nodes*  $N_F$ ,  $F = 1, K, 6$ , one node for each corner, called the *corner nodes*  $N_C$ ,  $C = 1, K, 12$ , and one node at the element centre, called the *central node*  $N_M$ . Thus, by including also the vertices, there are  $8 + 6 + 12 + 1 = 27$  *patch nodes*, involved in the splitting or un-splitting of a CUB8. These are set in an array  $P_p$ ,  $p = 1, K, 27$  in the following order, see Fig. 6: first the eight vertices of the parent element ( $P_{1\text{--}8} = V_{1\text{--}8}$ ), then the six face nodes ( $P_{9\text{--}14} = N_{F,1\text{--}6}$ ), then the twelve corner nodes ( $P_{15\text{--}26} = N_{C,1\text{--}12}$ ), and finally the central node ( $P_{27} = N_M$ ). The notation  $A_{a+b}$  is used as a shorthand to indicate all items from  $a$  to  $b$  in array  $A$ .

An important assumption in the algorithms to be presented below is that all descendent elements are *numbered consistently* with their parent. This means, for example, that if the first face of an element points (say) “downward”, then all its descendents’ first faces also point downward, *etc.* This choice is quite natural and greatly facilitates the splitting/un-splitting operations. However, this assumption can only be satisfied for elements which, in adaptivity, produce descendents of the same shape as—and which can be oriented in the same way as—their parent, like is the case here.

There are also other (constant) data useful in the element splitting/un-splitting process. Consider first the following problem: given a branch element  $i$ , identify the (corner) node  $N_c$  located in the middle of its  $c$ -th corner, *without* having at disposal the patch nodes table  $N_p$  for  $i$  (the construction of this table is relatively expensive and is performed only for the current element  $i$  being split or un-split, *not* for its adjacent elements). Note that node  $N_c$  does *not* belong to element  $i$ , but it belongs to at least one (more precisely, to two) of its descendents. A constant table  $C_c(m, c)$ ,  $m = 1, 2$ ,  $c = 1, K, 12$  solves the problem: the first entry  $C_c(1, c)$  is the index of the descendent, the second entry  $C_c(2, c)$  is the index of the node. Thus, for example, the corner node on corner 1 of a branch element is the second node ( $C_c(2, 1) = 2$ ) of its first descendent ( $C_c(1, 1) = 1$ ).

A similar constant table  $F_f(m, f)$ ,  $m = 1, 2$ ,  $f = 1, K, 6$  allows to find the face node on the  $f$ -th face of a branch element  $i$ . The first entry  $F_f(1, f)$  is the index of the descendent; the second entry  $F_f(2, f)$  is the index of the node. Thus, for example, the face node on face 1 of a branch element is the third node ( $F_f(2, 1) = 3$ ) of its first descendent ( $F_f(1, 1) = 1$ ).

Finally, a constant table  $C_d(m, c)$ ,  $m = 1, 2$ ,  $c = 1, K, 12$  lists the two descendents adjacent to each corner  $c$  of a branch element, in the same order as corner nodes are listed in table  $C_i$ . For example, the descendents adjacent to the first corner of a parent are (in this order)  $d_1$  and  $d_2$  (thus  $C_d(1, 1) = 1$  and  $C_d(2, 1) = 2$ ). Note that, with the numbering conventions assumed here for the CUB8, it is  $C_i = C_c = C_d$ , so only one of these three tables would suffice. However, the tables are kept distinct for generality in view of the application of these algorithms to other element shapes.

### 3.2. Some auxiliary procedures

There are some auxiliary procedures which are useful in the splitting/un-splitting algorithms to be detailed below. The first one computes the *nature of a face*  $F_k$  of an element  $i$ , i.e. whether the face is internal to the computational domain or external, i.e. on the boundary of the domain, based upon the neighbor and the p-neighbor at the face. The procedure (Algorithm A.1) is listed in the Appendix.

Another procedure (Algorithm A.2 in Appendix) computes the *nature of an external corner*  $C_i$  of a descendent element  $d_j$ , i.e. whether this corner lies on a base corner  $C_B$  or on a base face  $F_B$ . A criterion to find whether a corner is internal or external will be presented below in Section 3.3, based on the concept of corner star introduced there. Note that with the element splitting/un-splitting strategy used here an element corner—whatever the level of the element—is either (completely) internal to the domain or (completely) external, i.e. on the domain boundary. It is impossible to have partially internal, partially external corners.

### 3.3. Corner neighbors and corner star data structure

The nature of a corner node (hanging, b-hanging, non-hanging) depends upon the whole set of elements “adjacent” (in a broad sense) to the corner. We call such elements *corner neighbors* and the list of these elements the *corner star* of element  $i$  with respect to (i.e. around) its corner  $c$ , see Fig. 7. The number of neighbors to a corner is potentially unlimited for an unstructured mesh, but in practice there are between one (element  $i$  itself) and slightly more than four elements in a corner star. Values much larger than four are unlikely because the elements would be badly shaped. For a *regular* mesh of CUB8 there are *exactly* between one and four elements in each corner star.

As shown in Fig. 7, it is useful to distinguish between a *complete* star and an *incomplete* star. A star is complete if the space around the corner is entirely filled by elements (without “holes”), else the star is incomplete. One sees then immediately that an element’s *corner* is *external* if and only if its corner star is incomplete, else the corner is *internal*. This distinction is useful because Algorithm A.2 needs to be applied only to external corners. Note incidentally that the nature of a corner *cannot* be computed based only upon the nature of its nodes: in fact a corner whose nodes are all external can be internal. The same holds for faces.

The procedure to determine whether or not a star is complete is listed in the Appendix (see Algorithm A.3).

### 3.4. Computing a corner star

The calculation of the corner star of an element  $i$  around its corner  $c$  is the most CPU-expensive component of the present algorithms, because it must deal with a large variety of geometrical cases; see e.g. Fig. 7. Two different approaches have been tentatively considered.

The first one consists in performing a (fast) geometrical search in the vicinity of the corner under consideration, in order to locate all the potentially involved elements. To

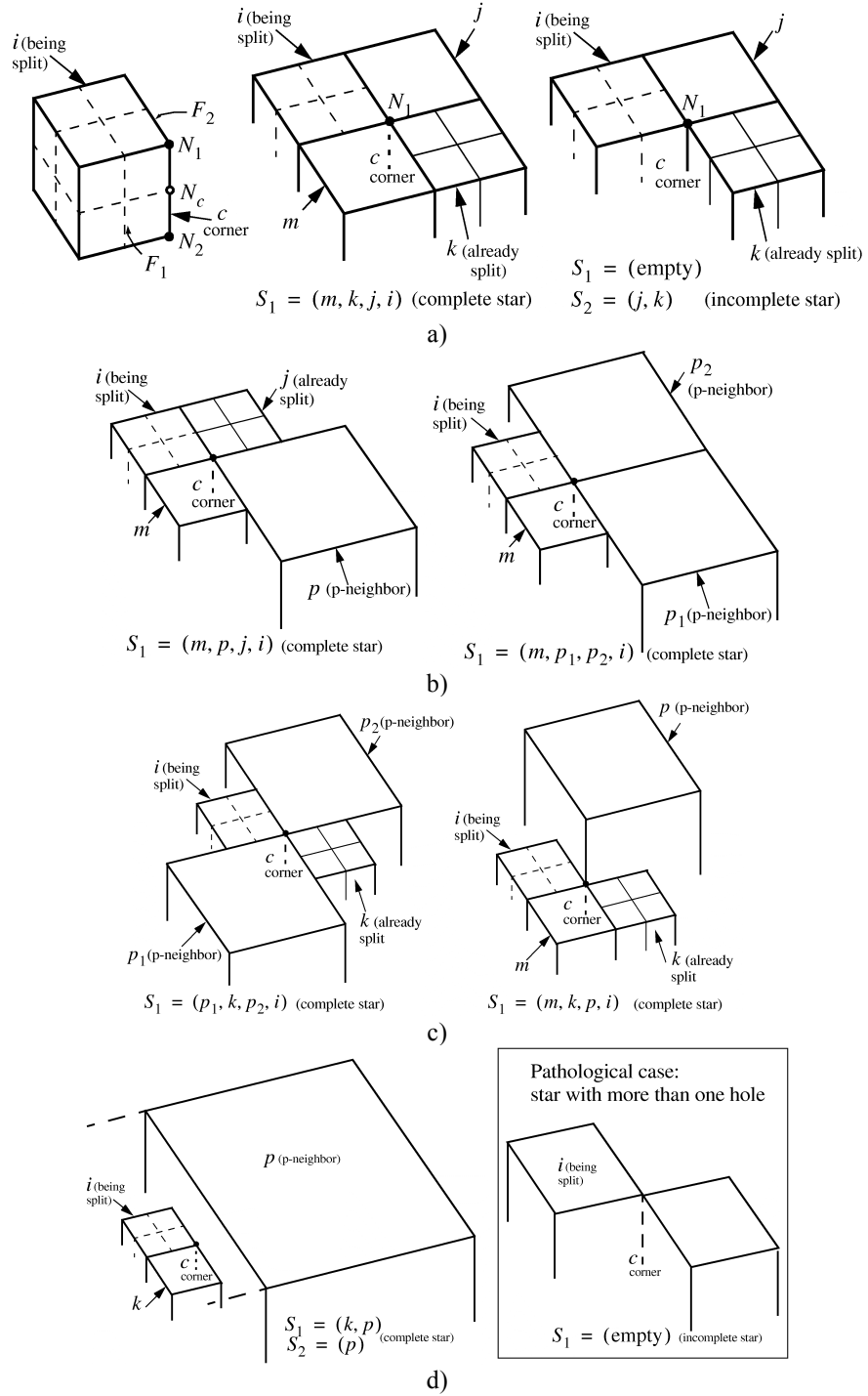


Fig. 7. Examples of corner stars (complete or incomplete).

this end the extent of the concerned zone around the corner has to be chosen, and this is not simple in the presence of elements of very different sizes (p-neighbors), like is the case here. The zone must be large enough to ensure that no potential neighbor is overlooked, but using large zones increases the cost of the search. A second drawback is that the search uses real-number quantities, e.g. nodal coordinates. The associated tests are delicate and the robustness of the method can be affected. Finally, elements which lie within the zone are not guaranteed to be corner neighbors and the verification requires further complex calculations.

The second approach is based on the observation that star elements are (recursively) the neighbors or p-neighbors of  $i$  across its two faces  $F_1, F_2$  adjacent to  $c$ , see Fig. 7a. So an algorithm can start from face  $F_1 = C_F(1, c)$  and recursively compute neighbor and p-neighbor elements (adjacent to this corner) until it either reaches (again) the current element  $i$ , or the search terminates. In the first case the star is complete, and this can be recognized by the presence of  $i$  as the last element in the list. In the second case, the same search process is repeated, but starting from face  $F_2 = C_F(2, c)$ . Thus a generic corner star is made of *two* element lists,  $S_1$  and  $S_2$ . Let  $n_{s_1}, n_{s_2}$  be the number of items in the lists. Note that element  $i$  is not inserted at the beginning of the lists, which start directly with  $i$ 's neighbor or p-neighbor across the corresponding face. Element  $i$  can be present (as the last item) in  $S_1$ , and in this case the star is complete just by using the first list, so that  $n_{s_2} = 0$  and  $S_2$  is not built up at all. However, element  $i$  cannot appear in  $S_2$ .

The advantage of the second method, which is the one chosen here, is that the search is entirely integer-based and therefore very robust. In fact, there are no tests on real-number quantities, subjected to a tolerance. Only element connectivity is used (nodal coordinates are never considered), and the knowledge of neighbors and p-neighbors resulting from the algorithms is fully exploited, resulting in a very efficient algorithm.

The method has only one limitation: since it is based on (recursively) using neighbors and p-neighbors, it can only deal with corner stars containing *at most one* "hole". As shown in Fig. 7d (last drawing), a star with more than one hole would not be entirely detected by this procedure (although it would be correctly marked as incomplete). However, such a case can be considered pathological, and thus impossible in real applications, since it is not good practice in Finite Elements to connect different 3D continuum mesh zones just along a corner.

The procedure to build up a corner star is relatively involved (although not difficult) due to the variety of possibilities, especially in the presence of p-neighbors, see some examples in Fig. 7. It is not completely detailed here for brevity. The constant data structures listed in Table 2 are exploited to speed up the calculations.

### 3.5. Element splitting algorithm

With the definitions and the procedures given in the previous Sections, the element splitting and un-splitting algorithms can now be detailed. The tasks of the element splitting algorithm are:

- To find the node numbers of the split element patch  $P_{1+27}$  defined in Section 3.1, by creating any face- or corner nodes as needed, or by locating and re-using them if they already exist.
- To define the hanging status (hanging, b-hanging or non-hanging) of newly created nodes; to check the hanging status of re-used nodes and to update it if necessary.

The algorithm consists of four parts:

- Loop on the eight parent element nodes to find the cube vertices  $P_{1+8}$ . Their hanging status is not affected.
- Loop on the six element faces to find the face nodes  $N_{9+14}$ .
- Loop on the twelve element corners to find the corner nodes  $N_{15+26}$ .
- Generate the element central node  $P_{27}$  and set it non-hanging.

Parts A) and D) of the algorithm are trivial and need no further comment. Parts B) and C) are detailed below. The updating of neighbors and p-neighbors just after an element is split is given in Section 3.8.

B) – Loop on element faces.

- Loop on the six faces  $F$  of current element  $i$ . Let  $j$  be the neighbor, and  $p$  the p-neighbor, of  $i$  across face  $F$ .
- If  $j = 0$ , i.e. there is no neighbor, then:
  - Create face node  $P_{8+F}$ .
  - If  $p = 0$  the face is external. Set  $P_{8+F}$  as b-hanging upon the four nodes of the corresponding *base face*  $F_B$ , i.e. face  $F$  of  $i$ 's *base element*.  $F_B$  must exist in this case, since  $i$  has neither a neighbor nor a p-neighbor across  $F$ .
  - Else  $p > 0$  and  $P_{8+F}$  is internal. Set it as hanging upon  $F$ 's four nodes.
  - Interpolate coordinates and nodal variables at  $P_{8+F}$ .
- Else  $j > 0$  (so it must be  $p = 0$ ).
  - If  $j$  is a leaf, then:
    - Create face node  $P_{8+F}$ .
    - Set it as hanging upon  $F$ 's four nodes.
    - Interpolate coordinates and nodal variables at  $P_{8+F}$ .
  - Else  $j$  is a branch. Then:
    - Find old face node (which must exist already)  $P_{8+F}$ .
    - Check that it was hanging and set it as non-hanging.
- Next face  $F$ .

C) – Loop on element corners.

- Loop on the twelve corners  $C$  of current element  $i$ , of level  $L$ . Let  $F_1, F_2$  be the two faces of  $i$  adjacent to the present corner  $C$ , let  $N_1, N_2$  be the end-nodes of corner  $C$ , and let  $N_C$  indicate the corner node (either existing or to be created), see Fig. 7a.
- Determine whether faces  $F_1, F_2$  are internal or external by Algorithm A.1.
- Build up the corner star of elements around  $C$ , see Section 3.4.

4. Find the corner node  $N_C$ , if it exists already, otherwise create it:
  - If the corner star is not empty and contains at least one element  $m$  (other than  $i$ ) of level  $L$  (i.e. if  $m$  has a corner  $C_m$  of extremes  $N_1, N_2$ ) and  $m$  is a branch, then  $N_C$  exists and is readily determined from  $m$ 's descendents. Check that  $N_C$  was hanging upon  $N_1$  and  $N_2$ . Set  $N_{14+C} = N_C$ .
  - Else  $N_C$  is created by interpolation between  $N_1$  and  $N_2$ . Set  $N_{14+C} = N_C$ .
5. Compute the (new) hanging status of  $N_C$ :
  - If the corner star is either empty or contains only elements of level  $L$  (i.e. if there are only neighbors and no p-neighbors) and if all such elements are branches, then:
    - If  $C$  is external (the star is incomplete), then  $N_C$  is b-hanging. More precisely:
      - If  $C$  is part of an (external) base corner  $C_B$  (see Algorithm A.2), then  $N_C$  b-hangs upon the two (base) nodes of  $C_B$ .
      - Else  $C$  is not part of a base corner, it just lies upon an (external) base face  $F_B^e$  (see Algorithm A.2), and  $N_C$  b-hangs upon the four (base) nodes of  $F_B^e$ .
    - Else  $C$  is internal (the star is complete) and  $N_C$  is non-hanging.
  - Else the corner star is not empty and either contains at least one element of level  $M < L$  (i.e. a p-neighbor), or it contains only elements of level  $L$ , but at least one of them is a leaf. Then,  $N_C$  is hanging upon  $N_1$  and  $N_2$ .
6. Next corner  $C$ .

### 3.6. Element un-splitting algorithm

The task of this algorithm is to un-split an element, i.e. to re-activate a previously split (idle) element, starting from its eight descendents (which must be all leaves). At the beginning of the algorithm  $P_{1+27} > 0$  and all corresponding nodes exist. Upon un-splitting some nodes are deleted, the others are kept but possibly their hanging status changes. The algorithm consists of four parts:

- A) Fill  $P_{1+27}$  from the descendents of the element being un-split.
- B) Loop on the six element faces to treat the face nodes  $P_{9+14}$ .
- C) Loop on the twelve element corners to treat the corner nodes  $P_{15+26}$ .
- D) Verify that the old central node  $P_{27}$  was non-hanging and destroy it.

Parts A) and D) of the algorithm are trivial and need no further comment. Parts B) and C) are detailed below. The updating of neighbors and p-neighbors after element un-splitting is given in Section 3.9.

B) – Loop on element faces.

1. Loop on six faces  $F$  of current element  $i$ . Let  $j$  be the neighbor to  $i$  across  $F$ .
2. If  $j = 0$ , i.e. if there is no neighbor, then:
  - Check that face node  $P_{8+F}$  was hanging or b-hanging, then destroy it.
3. Else  $j > 0$  i.e. there is a neighbor.
  - If  $j$  is a leaf: check that face node  $P_{8+F}$  was hanging, then destroy it.

- Else  $j$  is a branch. Check that face node  $P_{8+F}$  was non-hanging, then set it hanging upon the four face nodes of  $F$ .
4. Next face  $F$ .

C) – Loop on element corners.

1. Loop on the twelve corners  $C$  of current element  $i$ , of level  $L$ . Let  $N_1, N_2$  be the end-nodes of corner  $C$ , and let  $N_C$  indicate the corner node, see Fig. 7a.
2. Build up the corner star of elements around corner  $C$ , see Section 3.4.
3. If the corner star is not empty and contains at least one element  $m$  (other than  $i$ ) of level  $L$  (i.e.  $m$  has a corner  $C_m$  of extreme nodes  $N_1, N_2$ ), and  $m$  is a branch, then  $N_C$  is kept, and its hanging status is checked and set as follows:
  - If  $C$  is external, i.e. if the star is incomplete, then check that  $N_C$  was either b-hanging (either on two or on four base nodes) or hanging upon  $N_1, N_2$ .
  - Else  $C$  is internal, i.e. the star is complete. Check that  $N_C$  was either non-hanging or hanging upon  $N_1, N_2$ .
  - Set  $N_C$  hanging upon  $N_1$  and  $N_2$ .
4. Else, destroy  $N_C$ .
5. Next corner  $C$ .

### 3.7. The QUA4 quadrilateral

The element splitting and un-splitting algorithms for the QUA4 quadrilateral in 2D follow exactly the same strategy as those for the CUB8 element in 3D, with obvious adjustments in the number of nodes, faces, neighbors *etc.* The most notable simplification is the absence of corners, and therefore also of corner stars, so that parts C) of the algorithms of Sections 3.5 and 3.6 do not exist in 2D.

### 3.8. Treatment of $p$ -neighbors upon element splitting

When an element  $i$  of level  $L$  is split as shown in Section 3.5, it generates descendents of level  $L+1$  and  $i$  is flagged as idle. Then:

- Any element  $p$  that was a  $p$ -neighbor of  $i$  across a certain face, remains its  $p$ -neighbor across that same face, see Fig. 8a.
- Any element  $k$  which had  $i$  as a  $p$ -neighbor must be treated. Such elements are sought among *all* descendents (both leaves and branches) of  $i$ 's neighbors, at any level  $K > L$ . Note that  $i$ 's  $p$ -neighbors are not considered here because they are necessarily at a level  $M < L$  and by definition they have no descendents. Let  $K$  be the level of one such element  $k$ , which had  $i$  as  $p$ -neighbor. Then:
  - If  $K = L+1$ , then one of the descendents of  $i$  (to be determined) becomes neighbor of  $k$  and reciprocally, see Fig. 8b.
  - Else  $K > L+1$ . Then one of the descendents of  $i$  (to be determined) becomes  $p$ -neighbor of  $k$ , see Fig. 8c.



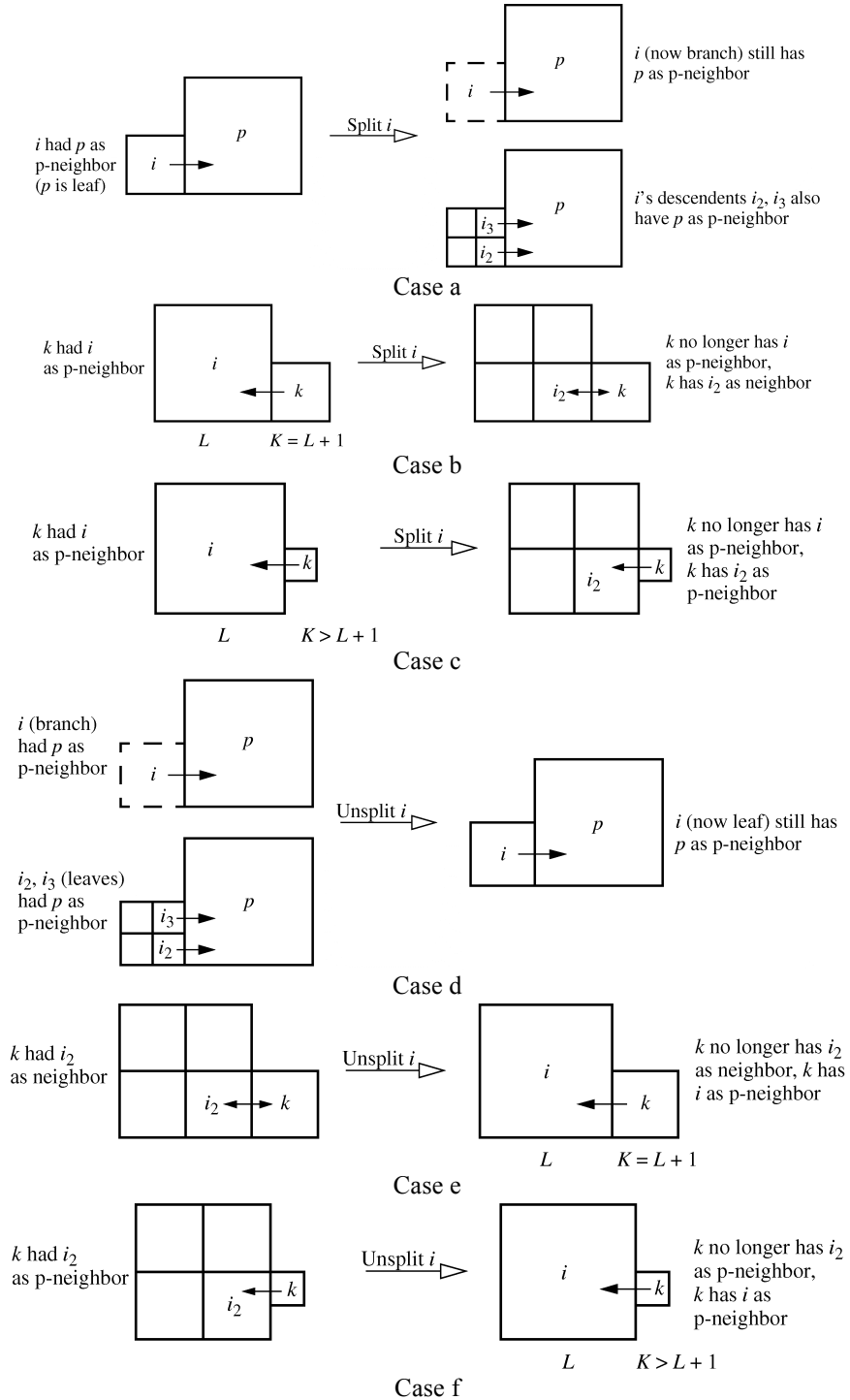


Fig. 8. Updating neighbors and p-neighbors.

### 3.9. Treatment of *p*-neighbors upon element un-splitting

When an element  $i$  of level  $L$  is un-split as shown in Section 3.6, its descendents of level  $L + 1$  are destroyed and  $i$  (which was flagged as idle) becomes active again. Then:

- Any element  $p$  that was a  $p$ -neighbor of  $i$  across a certain face, remains its  $p$ -neighbor across that same face, see Fig. 8d.
- No elements could have  $i$  as  $p$ -neighbor because  $i$  was idle (branch). However, some elements  $k$  could have one of  $i$ 's descendents as neighbor or  $p$ -neighbor. Such elements must be sought among *all* descendents (both leaves and branches) of  $i$ 's neighbors, at any level  $K > L$ . Note that  $i$ 's  $p$ -neighbors are not considered here because they are necessarily at a level  $M < L$  and by definition they are leaves. Let  $K$  be the level of one of such elements  $k$ , having one of  $i$ 's descendents as neighbor or  $p$ -neighbor. Then:
  - If  $K = L + 1$ , then  $k$  and one of  $i$ 's descendents (to be determined) were neighbors. Element  $i$  becomes  $p$ -neighbor of  $k$ , see Fig. 8e.
  - Else  $K > L + 1$ . Then  $k$  had one of  $i$ 's descendents (to be determined) as  $p$ -neighbor. Element  $i$  becomes  $p$ -neighbor of  $k$ , see Fig. 8f.

## 4. Exploiting the adaptive data structure

The knowledge of hanging and b-hanging nodes resulting from the algorithms of Section 3 is exploited in order to impose suitable constraints ensuring consistency of the adaptive solution. First, continuity of the solution must be satisfied at nodes on locally non-conforming element-to-element interfaces (hanging nodes). Second, essential boundary conditions at b-hanging nodes must be inherited from the corresponding (base) master nodes. All such constraints are of course non-permanent because nodes nature can change step by step during the transient.

Finally, the notion of neighbor and  $p$ -neighbor across an element's face allows to efficiently compute numerical fluxes (transport terms) in adaptive fluid calculations.

### 4.1. Constraints on hanging nodes

As an example of hanging node consider node  $h_1$  in the right part of Fig. 1, which results from the splitting of element  $i$ . In the current mesh configuration this node hangs upon two master nodes,  $K$  and  $L$ . These happen to be *base* nodes in the present case, but this is not necessary in general, and is irrelevant as concerns the proposed treatment.

A displacement-based Finite Element formulation is used in the code. Time integration is done explicitly by the Central Difference scheme and the fundamental quantity is the velocity  $\mathbf{v}$ , discretized at nodes. Therefore, in order to ensure continuity of the solution around a generic hanging node  $H$  the constraint to be imposed is:

$$\mathbf{v}_H = \sum_{i=1}^m N_i \mathbf{v}_{M_i}. \quad (1)$$

where  $M_i$  are the  $m$  master nodes upon which node  $H$  is hanging, and  $N_i$  are suitable coefficients. In the example of Fig. 1,  $m = 2$  and eq. (1) becomes  $\mathbf{v}_{h_1} = (\mathbf{v}_K + \mathbf{v}_L) / 2$ .

Constraints (1) are written for each hanging node. Each constraint is split into  $d$  components, one for each global axis, where  $d$  is the space dimension (2 or 3). All such constraints, plus any essential boundary conditions imposed by the user, form a linear system of constraints on the (velocity) degrees of freedom (dofs) of the system. Note that this system contains *only the constrained dofs*, not all system dofs. To enforce such constraints a method of Lagrange multipliers is used. This requires the numerical solution, at each time step, of a linear system of equations and is the only implicit part of the transient solution strategy. Interested readers can find full details of the procedure in references [Casadei *et al.* (1995, 2009)].

#### 4.2. Constraints on b-hanging nodes

As an example of b-hanging node consider node  $b_1$  in the right part of Fig. 1, which results from the splitting of element  $i$ . In the current mesh configuration this node b-hangs upon two master nodes,  $I$  and  $J$ . Note that, in contrast with the case of hanging nodes of Section 4.1, the masters of a b-hanging node are always base nodes.

An important practical aspect of using adaptivity in real applications is the specification of essential boundary conditions. Here we assume that users only know the base mesh, which is provided in input to the code, and therefore *boundary conditions are specified only for the base nodes*. It is then desirable that such conditions be automatically propagated to any descendent nodes on the boundary that are (automatically, i.e. out of user's control) created during the mesh adaptation process.

To this end, we exploit the knowledge of b-hanging nodes resulting from the algorithms of Section 3. The  $m$  masters  $M_i$  of a generic b-hanging node are inspected. If *all* of them share the *same* type of boundary condition, then this condition is imposed on the b-hanging node as well, and is added to the system of constraints to be solved by the Lagrange multipliers method as described in Section 4.1. This explains why the masters of a b-hanging node are always base nodes: because boundary conditions are explicitly known only for base nodes, not for descendents.

For example, assume that in the case of Fig. 1 node  $I$  is blocked in the vertical direction, while node  $J$  is blocked in both directions. Then, b-hanging node  $b_1$  would also be blocked in the vertical direction. This strategy works well, at least for the simplest types of boundary conditions, as shown in the numerical examples of Section 5.

#### 4.3. Numerical fluxes in fluid calculations

The knowledge of the neighbor or p-neighbor at each element face resulting from the algorithms of Section 3 allows a precise and efficient calculation of *numerical fluxes* across element-to-element interfaces, an essential ingredient in the solution of fluid equations. The procedure is briefly outlined for the case of fluid modeling by Finite Elements using a classical *fractional step* approach, but it can be extended along the same lines also to other schemes, e.g. to node-centered or cell-centered Finite Volume formulations.

In the chosen fractional step approach, transport terms (numerical fluxes) across neighboring elements—resulting from Euler equations for compressible inviscid fluids—are computed according to the so-called *lowest-index rule*.

Assuming for the moment a conforming mesh, if elements  $i$  and  $j$  are neighbors at a given face, then the flux of mass and energy across the face is evaluated (with the appropriate sign, depending on nodal velocities) while treating the element with the *minimum index*, i.e.  $\min(i, j)$  in the general loop over all elements. This ensures two things: first, the transport across each face is evaluated only once (correctness); second, when an element is treated all fluxes across its faces have been evaluated so that the element state can be directly updated, without the need of an additional loop over elements (efficiency). This algorithm is generalized as follows to the case of non-conforming meshes (adaptivity):

1. Set total mass and energy fluxes to zero for all elements.
2. Loop over elements. Let  $i$  be the current element.
3. Loop over  $i$ 's faces. Let  $F$  be the current face,  $j$  the neighbour and  $p$  the  $p$ -neighbor of  $i$  across face  $F$ .
4. If  $j = 0$  then:
  - If  $p = 0$  or  $0 < p < i$  then skip flux calculations for face  $F$ .
  - Else  $p > i$ . Compute the (signed) mass and energy fluxes from  $i$  to  $p$  across  $F$ , subtract them from the total fluxes of  $i$  and add them to the total fluxes of  $p$ .
5. Else  $j > 0$ . Then:
  - If  $j$  is a leaf, then:
    - If  $j < i$  then skip flux calculations for face  $F$ .
    - Else  $j > i$ . Compute the (signed) mass and energy fluxes from  $i$  to  $j$  across  $F$ , subtract them from the total fluxes of  $i$  and add them to the total fluxes of  $j$ .
  - Else  $j$  is a branch. Then loop on all active (leaf) descendents  $d_j$  of  $j$  having  $i$  as  $p$ -neighbor across one of their faces  $f_j$ :
    - If  $d_j < i$  then skip flux calculations for face  $f_j$ .
    - Else  $d_j > i$ . Compute the (signed) mass/energy fluxes from  $i$  to  $d_j$  across  $f_j$ , subtract them from the total fluxes of  $i$  and add them to the total fluxes of  $d_j$ . Note that the geometry of (the smaller) face  $f_j$ , and not of (the larger) face  $F$ , is used in this case to compute the fluxes.
    - Next  $d_j$ .
6. Next face (GOTO 3).
7. All faces have been considered for the current element  $i$  and therefore its total mass and energy fluxes have been computed. Update the element's physical state and compute internal forces.
8. Next element (GOTO 2).

## 5. Numerical examples

Three numerical examples are presented to illustrate the proposed mesh refinement and un-refinement algorithms in action. In all cases mesh adaptation is piloted by a special WAVE directive simulating the propagation of waves in a continuum. Two types of waves are considered in these tests: a plane wave and a spherical wave. The first type is characterized by a source point and by a direction of propagation, while the second only requires the source point. To each wave are assigned a constant imposed propagation speed  $v$  and a starting time  $t_0$ . Each wave front has two associated length parameters:  $h_1$  specifies the thickness of the wave front zone in which the mesh has to be refined up to an imposed maximum level  $L_{\max}$ ;  $h_2$  specifies the thickness of the whole wave. The mesh refinement level is varied linearly from  $L_{\max}$  (finest mesh) to 1 (base mesh) in the zone between  $h_1$  and  $h_2$ .

The values of all wave parameters are prescribed according to known analytical solutions for the simple academic problems chosen. The following tests cannot be considered real adaptive calculations, because in adaptivity mesh refinement should rather be (automatically) piloted by suitable error estimators/indicators. However, the tests are sufficient to check all geometric aspects of the proposed mesh refinement and un-refinement algorithms, and to verify their effects on explicit numerical solutions in fast transient dynamics.

### 5.1. Spherical wave in a 3D slab

The first test simulates propagation of a spherical wave in a square slab of  $10 \times 10$  units and of thickness 1. The wave originates in one corner of the slab at time 0. The base mesh consists of  $10 \times 10 = 100$  regular cubes and the chosen wave parameters are:  $L_{\max} = 3$ ,  $h_1 = 1.5$ ,  $h_2 = 5.0$ ,  $v = 5000$ . The material is linear elastic but material properties are irrelevant in this case because the wave is purely fictitious: no loading is applied and thus no stresses are generated.

The initial mesh is shown in Fig. 9a. Note that some refinement occurs near the wave origin (marked by a dot) already at the initial time, so that the wave is then properly captured. Since a spherical wave is used rather than a cylindrical one, mesh refinement is not uniform across the slab thickness. This is done on purpose in order to submit the splitting and un-splitting algorithms to a larger variety of cases than with a cylindrical wave. Figs. 9b and 9c show the advancing wave front at 1.5 ms on the surface and within the body, respectively. Note that with the chosen parameters mesh transition is quite sharp and some base elements ( $L = 1$ ) are adjacent to some maximum-refined elements ( $L = 3$ ). This is probably not a good choice in practical applications but again, it is used here just to show that the proposed algorithms are general and can deal with arbitrary level jumps between neighboring elements. An option in the code allows to automatically prescribe smooth mesh transitions, such that the level jump between any couple of neighboring elements is at most one. This is to say that *the index of irregularity* of the mesh is one, or that a *1-irregular mesh* is prescribed, following the terminology introduced by Demkowicz *et al.* [1989].

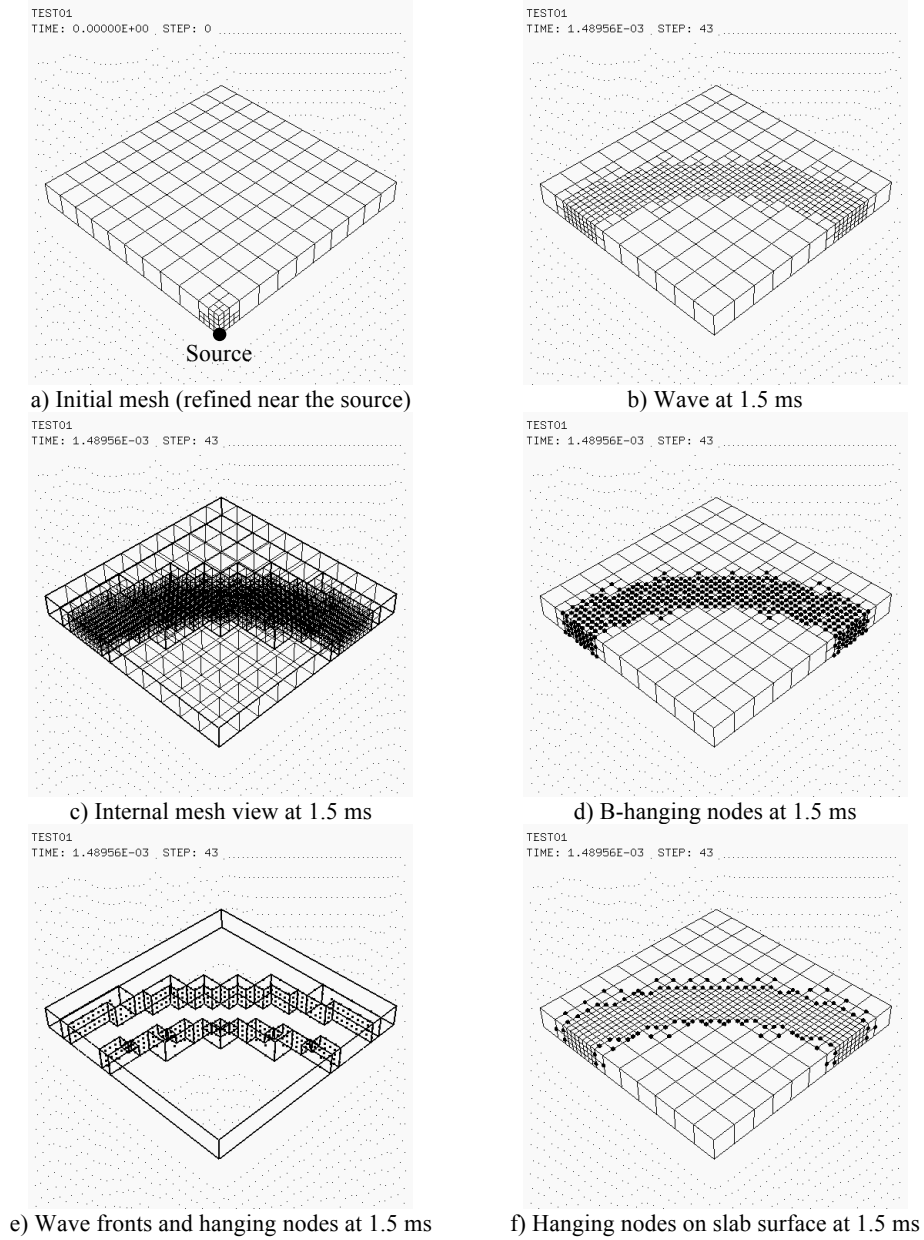


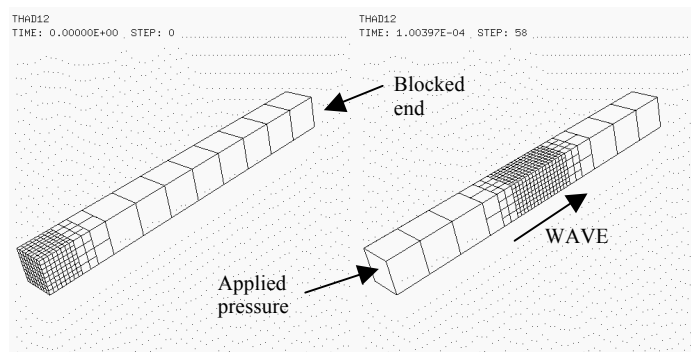
Fig. 9. Spherical wave propagation in a 3D slab.

Fig. 9d shows b-hanging nodes at 1.5 ms. These are all located on the slab surface, by definition. Finally, Figs. 9e and 9f show hanging nodes at 1.5 ms. These are located on the advancing wave fronts. In Fig. 9e the slab is made transparent to show all hanging nodes, most of which are in the body interior. Fig. 9f shows only the hanging nodes on the body surface, a possibility which exists only in 3D cases as mentioned above.

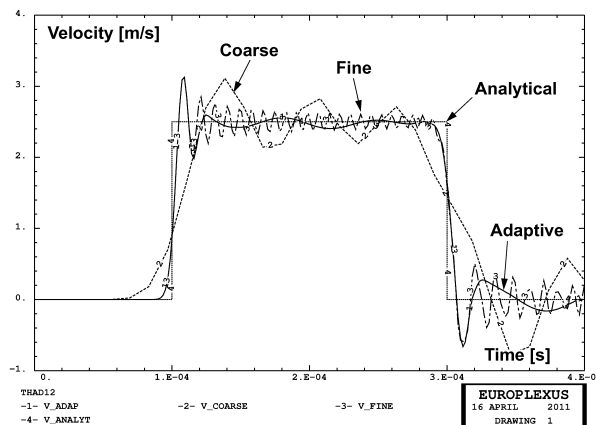
The mesh adaptation algorithms behave as expected in this test. The maximum number of elements reached during the transient (including both branches and leaves) is 2460, of which 100 are base elements. The maximum number of nodes is 3183, of which 242 are base nodes.

**5.2. Plane step wave in an elastic bar**

The second test considers an elastic bar of length  $l = 1$  m, with a square cross-section of  $0.1 \times 0.1$  m subjected to a constant pressure  $p = 1.0 \times 10^8$  Pa at the left end, and blocked at the right end. The material is linear elastic with density  $\rho = 8000$  kg/m<sup>3</sup>, Young's modulus  $E = 2.0 \times 10^{11}$  Pa, Poisson's coefficient  $\nu = 0$  (so that the problem is physically 1D). With these values, sound speed is  $c = \sqrt{E/\rho} = 5000$  m/s. Adaptivity is piloted by two WAVE directives. The first one represents the incident wave produced by the pressure load, starting at the left end at  $t = 0$  and propagating to the right. The second one represents the reflected wave, starting at the (blocked) right end at  $t = l/c = 2 \times 10^{-4}$  s and moving to the left. Both waves have  $h_1 = 0.15$ ,  $h_2 = 0.5$  and  $L_{max} = 4$ . The base mesh uses only 10 regular cubes.



a) Initial mesh and adapted mesh at 10  $\mu$ s



b) Comparison of velocities at bar mid-point

Fig. 10. Plane step wave in an elastic bar.

This test is a first example of the treatment of boundary conditions with the present adaptivity strategy. The pressure (an example of *natural* condition) is applied by a special *boundary-condition* (b.c.) *element* attached to the left end of the bar. This element has the shape of a 4-node quadrilateral, whose nodes are merged with the nodes of the (base) cube face at the left end of the bar. The advantage of this technique is evident from the mesh at  $t = 0$ , shown in Fig. 10a (first picture): the first WAVE command refines the bar mesh (cube elements) at the left end at  $t = 0$ , in order to properly capture the incoming wave. Whenever a cube is refined or un-refined, the algorithm checks whether there is a b.c. element attached to any of its (external) faces, and if so then the b.c. element is automatically split or un-split as well. In this way, the applied load (pressure in this case) is transferred from the parent to the descendent b.c. elements and, ultimately, properly scaled loads result on the appropriate surface nodes of the descendent cubes, in a fully automatic and transparent way.

The imposed blockage of nodes at the right bar end is another example of boundary condition, in this case of the *essential* type. It seems natural that users impose constraints only on the *base* nodes, in this case the four nodes of the right-most cube face. The present adaptivity strategy makes it relatively simple to program automatic transfer of constraints to any descendent nodes created in the adaptive process (Section 4.2). In fact, all such nodes are *b-hanging nodes* whose masters are the blocked face nodes. Since all masters are subjected to the *same* constraint (horizontal blockage), this constraint can be easily and automatically propagated to all the relevant descendent nodes. Fig. 10a (second picture) shows the adapted mesh at a later time, when the incident wave is traversing the bar. Note that the mesh at the left bar extremity (including the b.c. elements) has been automatically un-refined and the base mesh is recovered.

Fig. 10b compares solutions without and with adaptivity against the analytical solution of the bar problem. The curves represent the time history of velocity at the bar center. The analytical solution is the step function represented by the dotted curve. Numerical solutions present oscillations, due to the elastic nature of the material (no numerical damping). Two solutions with uniform cube meshes are shown (dashed lines): one with a coarse mesh (only 10 elements) and one with a fine mesh (80 elements), corresponding to the maximum mesh refinement in the adaptive solution (level 4). The adaptive solution, represented by the solid line, coincides exactly with the fine-mesh solution near the jumps at 0.1 and 0.3 ms (where the WAVE directive keeps the mesh fine), while far from them it has an oscillatory behavior. Oscillations have lower amplitude and lower frequency than both uniform mesh solutions, so the adaptive solution looks somewhat smoother. Probably some numerical damping is introduced by the mesh un-refinement process, whereby stresses in a parent element are computed by averaging the stresses in its children. The adaptive solution looks very good: it captures the shocks as precisely as the fine-mesh model, and presents less oscillations.

The maximum number of cube elements reached during the transient (including both branches and leaves) is 1386, of which 10 are base elements. The maximum number of special b.c. elements (to impose the pressure) reached during the transient (including both branches and leaves) is 85, of which only 1 is a base element. The maximum number of



nodes is 1656, of which 44 are base nodes. The same bar problem has been solved also with 2D elements, QUA4 4-node quadrilaterals. Results (not shown for brevity) are nearly identical to the 3D case, both with uniform and with adaptive meshes.

### 5.3. Shock tube

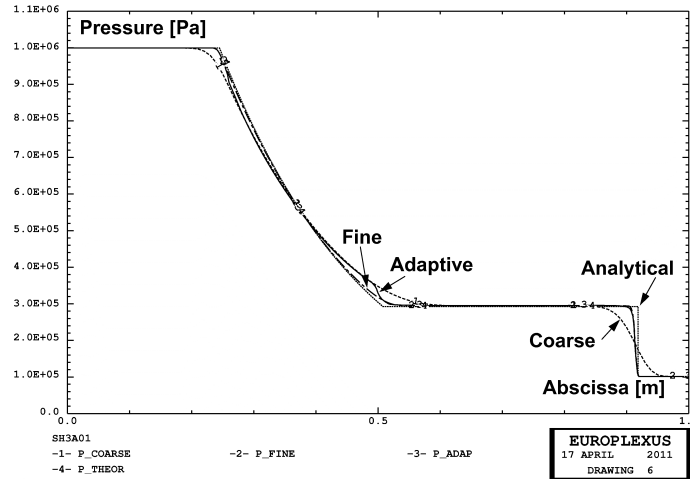
The third and last test is the classical shock tube problem. A rigid tube of length  $l = 1$  m and  $0.01 \times 0.01$  m square cross section is filled by a perfect gas and is subdivided in two equal parts by an ideal wall. The left part is initially at higher pressure than the right part. At the initial time the separation between the two parts is removed and waves start to propagate along the tube: a *shock wave* and a *contact discontinuity* wave propagate towards the low-pressure zone, and a *rarefaction wave* propagates towards the high-pressure zone. A complete analytical solution of this problem is available.

The assumed gas equation is  $p = (g - 1)ri$  where  $p$  is the pressure,  $g$  is the ratio of specific heats,  $r$  is the density and  $i$  is the specific internal energy. We take  $g = 1.269$  and  $i = 3.046 \times 10^6$  J/kg in both zones. The left zone has  $r_1 = 1.22$  kg/m<sup>3</sup> and thus  $p_1 = 1 \times 10^6$  Pa, while the right zone has  $r_2 = 0.1237$  kg/m<sup>3</sup> and thus  $p_1 = 1.01 \times 10^5$  Pa.

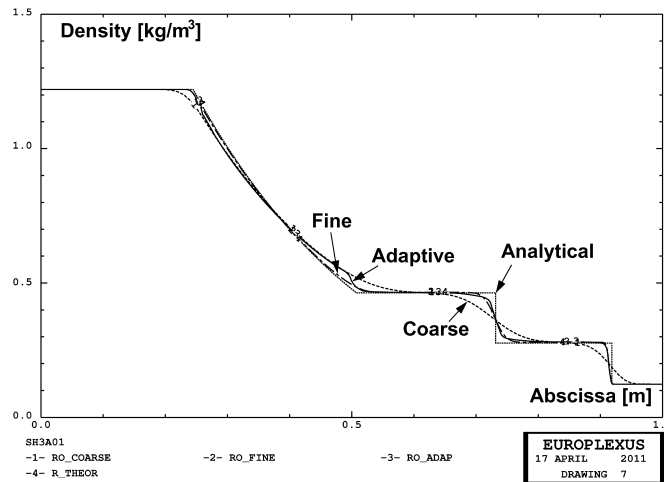
Two uniform-mesh solutions are obtained, one with a coarse mesh of 100 cube fluid elements and the other with a fine mesh of 800 cubes. Then an adaptive solution is obtained using a base mesh of 100 cubes, and *four* WAVE directives, one for the shock wave, one for the contact discontinuity and two for the initial and final fronts of the rarefaction fan. All waves originate at the tube center at the initial time and propagate in the relevant direction with the analytically computed velocities: 1672 m/s for the shock, 925.4 m/s for the contact discontinuity, - 30.12 m/s and - 1020 m/s for the rarefaction wave. All waves use  $h_1 = 0.015$ ,  $h_2 = 0.05$  and  $L_{\max} = 4$ . The boundary conditions are as follows: all (base) nodes on the tube surface are blocked in the  $y$  and  $z$  directions; the four (base) nodes of the left tube face and the four (base) nodes of the right tube face are blocked also in the  $x$  direction. Like in the previous example, b-hanging (descendent) nodes automatically inherit such constraints from the corresponding master (base) nodes, thanks to the strategy proposed in Section 4.2. Numerical fluxes in the fluid are computed according to the technique described in Section 4.3.

Figs. 11a and 11b show the distributions of fluid pressure and of fluid density along the tube, respectively, at 0.25 ms. The analytical solution is the step function represented by the dotted curve. The two solutions with uniform (coarse or fine) meshes are the dashed curves. The adaptive solution is the solid line. Both the shock and the contact discontinuity are captured by the adaptive solution with the same accuracy as the fine-mesh solution. In the rarefaction wave, only the two fronts are captured with great accuracy, while inside the fan the adaptive solution is similar to the coarse-mesh solution. This is normal since the chosen WAVE directives refine the mesh only at the fronts. All solutions are quite smooth and no oscillations are induced by mesh adaptation.

The maximum number of elements reached during the transient (including both branches and leaves) is 5084, of which 100 are base elements. The maximum number of



a) Pressure along the tube at 0.25 ms



b) Density along the tube at 0.25 ms

Fig. 11. Shock tube.

nodes is 6290, of which 404 are base nodes. Identical uniform-mesh and adaptive solutions are obtained in 2D with quadrilateral elements, and are not presented here for brevity.

#### 5.4. Efficiency

A rough estimate of the efficiency of the proposed mesh refinement and un-refinement algorithms can be obtained by comparing CPU times for uniform-mesh and adaptive solutions. Such times are too small for the elastic bar test of Section 5.2, but for the shock tube problem of Section 5.3 (3D version) we have the following results. The adaptive solution with 100 base elements, 4 WAVE directives and  $L_{\max} = 4$  needs 44 s on a laptop

computer. Since the present algorithms refine the mesh in all spatial directions, this should be compared against a uniform fine-mesh non-adaptive solution with  $8 \times 8 \times 800 = 51200$  elements, which needs 229 s. Therefore, a speed-up factor of 5.2 is obtained in this case (including the overhead needed to compute the WAVE fronts in the adaptive solution).

## 6. Conclusions and perspectives

The paper presents procedures to arbitrarily refine and un-refine a computational grid of QUA4 (in 2D) or CUB8 (in 3D) element shapes. The chosen strategy, based only upon element connectivity (integer data), is simple and robust and lends itself well to fast transient dynamic applications, dominated by wave propagation.

The numerical tests, performed using a simple wave propagation paradigm (WAVE directive) both in solid- and in fluid mechanics, show that mesh-adaptive solutions are as accurate as uniform fine-mesh solutions near the advancing wave fronts, without causing instability or loss of accuracy in zones where the solution is smooth.

Special attention is devoted to boundary conditions in adaptivity, an aspect of great importance in realistic applications. As concerns essential conditions, a technique exploiting the information resulting from the proposed mesh adaptation algorithms allows propagating the user-imposed constraints from the base nodes to the descendent (adaptive) nodes in an automatic and transparent way. Similarly, for natural conditions (e.g. an imposed pressure) a technique, based on special boundary-condition elements also subjected to adaptivity (in a natural way), is proposed in the bar test of Section 5.2.

This is only a first step (covering mostly geometric aspects) towards implementation of full mesh adaptivity in fast dynamics. Ongoing work focuses on error indicators, which should ultimately be used to automatically pilot mesh adaptation especially in fast transient fluid-structure interaction problems, see reference [Casadei *et al.* (2011)], by the algorithms proposed in this paper. To this end, calculation of numerical fluxes in adaptive fluid meshes will have to be extended to Finite Volume formulations, along the lines already presented for Finite Elements in Section 4.3.

## References

- Burstedde, C., *et al.* (2009). ALPS: a framework for parallel adaptive PDE solution. *Journal of Physics: Conference Series* 180, doi 10.1088/1742-6596/180/1/012009.
- Casadei, F., Halleux, J.P. (1995). An algorithm for permanent fluid-structure interaction in explicit transient dynamics. *Computer Methods in Applied Mechanics and Engineering*, **128**(3-4): 231–289.
- Casadei, F., Halleux, J.P. (2009). Binary spatial partitioning of the central-difference time integration scheme for explicit fast transient dynamics. *International Journal for Numerical Methods in Engineering*, **78**: 1436–1473.
- Casadei, F., Larcher, M. and Leconte, N. (2011). Strong and weak forms of fully non-conforming FSI algorithm in fast transient dynamics for blast loading of structures. *Proceedings of the COMPDYN 2011 Conference*, Papadrakakis, M., Fragiadakis, M., Plevris, V. (eds.), Corfu, Greece, 26–28 May 2011.

- Casadei, F., *et al.* (2012). EUROPLEXUS User's Manual: see <http://europlexus.jrc.ec.europa.eu/>.
- Demkowicz, L., Devloo, Ph. and Oden, J.T. (1985). On an h-type mesh-refinement strategy based on minimization of interpolation errors. *Computer Methods in Applied Mechanics and Engineering*, **53**: 67–89.
- Demkowicz, L., *et al.* (1989). Towards a universal h-p adaptive Finite Element strategy, Part 1. Constrained approximation and data structure. *Computer Methods in Applied Mechanics and Engineering*, **77**: 79–112.
- Erhart, T., Wall, W.A. and Ramm, E. (2006). Robust adaptive remeshing strategy for large deformation, transient impact simulations. *International Journal for Numerical Methods in Engineering*, **65**: 2139–2166.
- Frey, P.J. and Alauzet, F. (2005). Anisotropic mesh adaptation for CFD computations. *Computer Methods in Applied Mechanics and Engineering*, **194**: 5068–5082.
- Kee, Bernard B.T., Liu, G.R. and Lu, C. (2008). A least-square radial point collocation method for adaptive analysis in linear elasticity. *Engineering Analysis with Boundary Elements*, **32**: 440–460.
- Li, Y. and Liu, G.R. (2011). An adaptive NS/ES-FEM approach for 2D contact problems using triangular elements. *Finite Elements in Analysis and Design*, **47**: 256–275.
- Liu, G.R. and Tu, Z.H. (2002). An adaptive procedure based on background cells for meshless methods. *Computer Methods in Applied Mechanics and Engineering*, **191**: 1923–1943.
- Liu, G.R., Kee, Bernard B.T. and Chun, L. (2006). A stabilized least-squares radial point collocation method (LS-RPCM) for adaptive analysis. *Computer Methods in Applied Mechanics and Engineering*, **195**: 4843–4861.
- Meyer, A. (2009). Error estimators and the adaptive Finite Element method on large strain deformation problems. *Mathematical Methods in the Applied Sciences*, **32**: 2148–2159.
- Nguyen-Thoi, T., Liu, G.R. *et al.* (2009). Adaptive analysis using the node-based smoothed finite element method. *Communications in Numerical Methods in Engineering*, DOI: 10.1002/cnm.1291.
- Nithiarasu, P. and Zienkiewicz, O.C. (2000). Adaptive mesh generation for fluid mechanics problems. *International Journal for Numerical Methods in Engineering*, **47**: 629–662.
- Peraire, J., *et al.* (1987). Adaptive remeshing for compressible flow computations. *Journal of Computational Physics*, **72**: 449–466.
- Tang, Q., Zhang, G.Y., Liu, G.R. *et al.* (2011). A three-dimensional adaptive analysis using the meshfree node-based smoothed point interpolation method (NS-PIM). *Engineering Analysis with Boundary Elements*, **35**: 1123–1135.
- Xu, George X., Liu, G.R. and Tani, A. (2010). An adaptive gradient smoothing method (GSM) for fluid dynamics problems. *International Journal for Numerical Methods in Fluids*, **62**: 499–529.
- Yerry, M.A. and Shephard, M.S. (1983). A modified quad-tree approach to Finite Element mesh generation. *IEEE Computer Graphics and Applications*, **3**(1): 34–46.
- Zhang, G.Y., Liu, G.R., and Li, Y. (2008). An efficient adaptive analysis procedure for certified solutions with exact bounds of strain energy for elasticity problems. *Finite Elements in Analysis and Design*, **44**: 831–841.

Zhang, J., Liu, G.R. *et al.* (2008). A gradient smoothing method (GSM) based on strong form governing equation for adaptive analysis of solid mechanics problems. *Finite Elements in Analysis and Design*, **44**: 889–909.

## Appendix

Here are the auxiliary procedures mentioned in Sections 3.2 and 3.3.

### *Algorithm A.1* - Determination of internal and external faces.

Let  $j$  be the neighbor and  $p$  the  $p$ -neighbor of element  $i$  across its face  $F_k$ . Then:

- If  $j > 0$ , then  $F_k$  is internal and lies upon a locally conforming element-to-element interface.
- Else, if  $p > 0$  then  $F_k$  is internal and lies upon a locally non-conforming element-to-element interface.
- Else,  $F_k$  is external.

### *Algorithm A.2* - Determination of base corner or base face of an external corner.

Let  $c$  be an *external* corner of element  $i$  (see criterion in Section 3.3). Let  $F_1 = C_f(1, c)$  and  $F_2 = C_f(2, c)$  be the two faces of  $i$  adjacent to  $c$ , see Table 2. Then:

- If  $F_1$  and  $F_2$  are either both external or both internal (see Algorithm A.1), then corner  $c$  lies upon a base corner  $c_B$ . This is the  $c$ -th corner of  $B_i$ , the base element from which element  $i$  descends (this element is called the *base ancestor* of  $i$ ). The base ancestor is readily determined by (recursively) computing  $i$ 's parent up to level 1 in the elements tree.
- Else,  $F_1$  and  $F_2$  are one external ( $F_e$ ) and one internal ( $F_i$ ). Let  $d_A, d_B$  be the two descendents of the generic CUB8 element adjacent to its  $c$ -th corner:  $d_A = C_D(1, c)$  and  $d_B = C_D(2, c)$ , see Table 2. Then:
  - If  $i$  is either the  $d_A$ -th or the  $d_B$ -th descendent of its parent, and if this property holds recursively up to level 1 of the elements tree, then corner  $c$  lies upon a base corner  $c_B$ , namely the  $c$ -th corner of  $B_i$ , the base ancestor of  $i$ .
  - Else, corner  $c$  lies upon a base face  $F_B$ , namely the  $F_e$ -th face of  $B_i$ , the base ancestor of  $i$ .

### *Algorithm A.3* - Determination of complete or incomplete corner star.

The corner star around corner  $c$  of element  $i$  (see Section 3.4), represented by element lists  $S_1$  and  $S_2$ , of length  $n_{S_1}, n_{S_2}$  is complete if and only if:

- Either  $n_{S_1} > 0$  and  $S_1(n_{S_1}) = i$ . In this case it is  $n_{S_2} = 0$ .
- Or,  $n_{S_1} > 0$  and  $n_{S_2} > 0$  and  $S_1(n_{S_1}) = S_2(n_{S_2}) \pi i$ . This happens when there is a “big”  $p$ -neighbor without a corner superposed to corner  $c$  which “closes” the star, see e.g. Fig. 7d. We denote such a  $p$ -neighbor a *face  $p$ -neighbor*, as opposed to a *corner  $p$ -neighbor*.