Design and Implementation of the Three-Dimensional Autonomous Leaves Graph Data Structure

MAYCON DA COSTA, DENISE BURGARELLI, RODNEY JOSUÉ BIEZUNER, RAFAEL SACHETTO OLIVEIRA and WAGNER MEIRA JR., Universidade Federal de Minas Gerais
RODRIGO WEBER DOS SANTOS, Universidade Federal de Juiz de Fora

Autonomous Leaves Graph (ALG), a dynamic space-time data structure designed to minimize the computational effort of refinement and derefinement operations and to allow flexibility in the levels of refinement of adjacent regions in space adaptive meshes, was introduced in [Burgarelli 1998] and implemented for two-dimensional domains. Its storage requirements and computational cost compare competitively with other mesh refinement schemes based on hierarchical trees. Here an algorithm for implementing this scheme for time-evolving three-dimensional spatial domains (ALG-3D) is given in full detail.

Categories and Subject Descriptors: E.1 [**Data Structures**]: Graphs and networks

General Terms: Algorithms, Design

Additional Key Words and Phrases: adaptive mesh refinement, ALG data structure, three-dimensional domains, space filling curve

1. INTRODUCTION

The computational geometric representation of time-evolving spatial data, as in Evolutionary Partial Differential Equations (PDEs), Geographic Information Systems (GIS) and dynamic image processing require efficient data structures to handle the ever changing adjacency relations between cells, the primary units of data. The

This work was partially funded by CNPq, Capes, Fapemig and InWeb.

Rafael Sachetto Oliveira is also affiliated with Universidade Federal de São João del Rei.
Authors' addresses: D. Burgarelli and R. J. Biezuner, Departamento de Matemática, ICEx, Universidade Federal de Minas Gerais, Av. Antônio Carlos 6627, Pampulha, CEP 30161-970, Belo Horizonte, MG, Brazil. emails: burgarelli@ufmg.br, rodney@mat.ufmg.br. R.S. Oliveira, Departamento de Ciência da Computação, Universidade Federal de São João del-Rei, CTAN, Prédio dos Cursos de Ciência da Computação e Zootecnia, 3 andar, Av. Visconde do Rio Preto, s/n, Colônia do Bengo, São João del-Rei, MG, Brazil, CEP 36301-360. email: sachetto@ufsj.edu.br. W. Meira Jr. and M. Costa, Departamento de Ciência da Computação, ICEx, Universidade Federal de Minas Gerais, Av. Antônio Carlos 6627, Pampulha, CEP 31270-010, Belo Horizonte, MG, Brazil. emails: mayconufmg@gmail.com , meira@dcc.ufmg.br. Rodrigo Weber dos Santos, Universidade Federal de Juiz de Fora, Instituto de Ciências Exatas - Prédio MMC - Campus Universitário, Juiz de Fora, MG, Brazil, CEP: 36036-330. email: rodrigo.weber@ufjf.edu.br.

goal is to minimize the number of operations of memory manipulation necessary to update the neighborhood structure. The level of cell refinement must be high enough to capture all the essential details but not unnecessarily refined, thus storing redundant information which may be excessively expensive to process. This can be achieved through adaptive mesh refining techniques, locally increasing the level of refinement when the data is more heterogeneous. As the data change with time, the system must be also capable of derefining, i.e. grouping again cells where the scenario appears more uniform.

In [Burgarelli 1998] the Autonomous Leaves Graph (ALG), an efficient graph-based data structure was introduced in order to handle the communication of cells in discretized domains to numerically solve evolutionary PDE's (see [Burgarelli et al. 2006]). The new data structure was favorably compared to commonly used tree-based data structures (quad-trees). The corresponding processing time spent in the communication between neighbor cells is independent of the number of cells present in the discretization (i.e., $O(1)$ for each cell). Although in [Burgarelli et al. 2006] ALG was applied only to the numerical solution of evolutionary PDE's in two-dimensional domains, neither of these restrictions apply to the data structure itself. ALG can be used in any type of problem where a geometrical domain with any number of dimensions is discretized. In the three dimensions of space, the presence of complex geometries with irregular boundaries introduces a whole new set of interesting phenomena which cannot be fully understood in a two dimensional spatial setting; thus there is a remarkable gain of insight when treating physical problems where all three spatial dimensions are considered, e. g. in computational fluid dynamics and computational geometry,(see [Gamezo et al. 2005], [Ji et al. 2008], [Penner et al. 2007] and [Tavakoli 2008]).

In this paper, we implement ALG in time-evolving three-dimensional spatial domains. We call this version ALG-3D. The algorithm is presented in all its details for the unit cube discretized through cubic cells. This domain was chosen in order to simplify the presentation, and since it is a domain of choice in many applications. The advantages of the previous ALG-2D structure over the simple two-dimensional quadtree structure are maintained when we move to the ALG-3D structure, namely its reduced time for communication between neighbor cells and the flexibility for refining and derefining dynamically the mesh structure, in comparison with simple three-dimensional octree structures.

The adaptation of the algorithm to regions of different shapes, or of different discretizations (like tetrahedra), although not straightforward, should be easier once the algorithm presented in this paper is fully understood.

The paper is arranged as follows. The data structure is discussed in Section 2. The refinement and derefinement techniques are presented in Sections 3 and 4 respectively. A node ordering algorithm based on a modified Hilbert curve is shown in Section 5. Finally, in Section 6 we present an application in cardiac electrophysiology.

2. DATA STRUCTURE

ALG-3D is a graph-based data structure used to represent a three-dimensional domain which supports adaptive refinement. For ease of presentation, the implementation in this paper is based on a unit cube, although any arbitrary solid region could be used.

2.1 Concepts

Start with a unit cube centered at the point $(1/2, 1/2, 1/2)$, so that it is entirely contained in the first octant of the standard xyz coordinate system. Create an initial mesh by uniformly dividing this cube into eight smaller cubic cells of length $1/2$. These eight cells are centered at the points $(1/4, 1/4, 1/4)$, $(1/4, 1/4, 3/4)$, $(1/4, 3/4, 1/4)$, $(1/4, 3/4, 3/4)$, $(3/4, 3/4, 1/4)$, $(3/4, 3/4, 3/4)$, $(3/4, 1/4, 1/4)$ and $(3/4, 1/4, 3/4)$, as shown in Figure 1(b). Each of these initial eight cells is identified according to the position it occupies in the mesh. The cell in the upper right corner of the frontal face of the mesh (meaning the face of the cube pointing in the positive x-direction), whose center is $(3/4, 3/4, 3/4)$, is called the front northeast cell. The remaining cells in the frontal face are, counterclockwise: front northwest, front southwest and front southeast. The cells of the back face of the mesh (closer to the yz plane) are, counterclockwise: back northeast, back northwest, back southwest and back southeast.

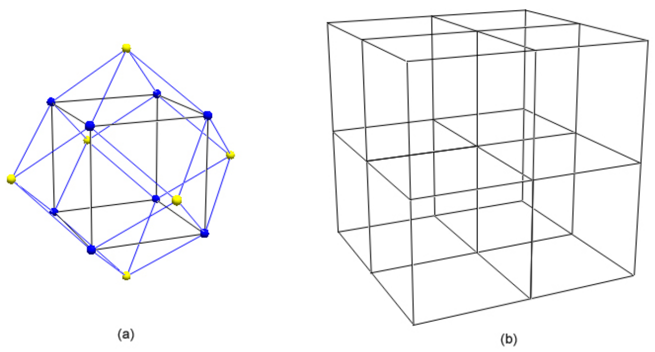


Fig. 1. Initial mesh and its associated graph.

Associated to this initial division of the unit cube into eight equal cubic cells is an initial graph (Figure 1(a)). To each cell corresponds a *cell node* in the graph (dark colored in Figure 1(a)). Each cell node stores any information associated to the cell it represents in variables, such as the spatial coordinates of its centers and physical states (if the numerical simulation of a physical phenomenon is intended; their number varies according to the problem being considered). Each of the eight cell nodes in Figure 1(a) has six links oriented along the six directions: *east*, along positive y direction; *west*, along negative y direction; *north*, along positive z direction; *south*, along negative z direction; *front*, along positive x direction, and *back*, along negative x direction. In the computer code, these six links are six pointers.

Each cell node has also two special pointers called *next* and *previous*. These pointers are used for ordering the cell nodes in the graph. Since the graph can become very irregular after successive refinements of different depths in different regions of the domain, it can become very difficult to visit the cells in a systematic way. This auxiliary chain list facilitates the access to the graph cell nodes, independently of any topological configuration the graph may assume. The maintenance of this list and the definition of the order of each cell node in it is done by the Hilbert curve, as will be explained in detail in Section 5.

The cell nodes also possess a variable that stores its current refinement level. When a cell node of level n is refined, its eight daughter cells will each have level $n + 1$. After a derefinement, the opposite happens: eight cells of level n fuse in order to constitute a cell of level $n - 1$.

In the graph structure underlying ALG there is another type of node called a *transition node*, which is used to connect cell nodes of different refinement levels. Each transition node has five links: one single connector and four quadruple connectors. These five links are implemented as five pointers in the computer code. They too are assigned variables corresponding to their refinement levels. In order to better understand the purpose of these transition nodes, consider the graph in Figure 2, which has a bunch of eight cells with one level of refinement higher than the refinement level of the initial cells.

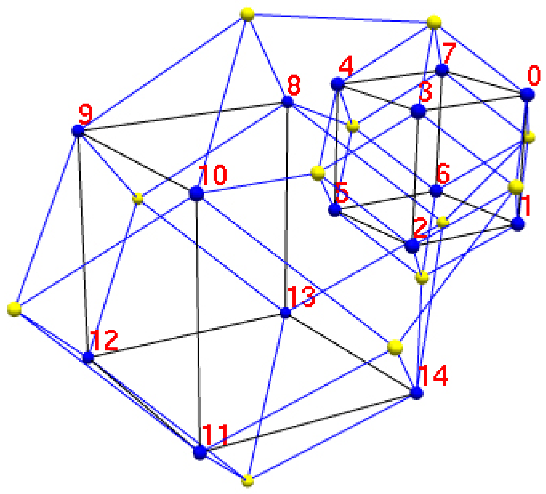


Fig. 2. Communication between cell nodes and transition nodes.

Cell nodes 2, 3, 4 and 5 have cell node 10 as the neighbor in the west direction. Reciprocally, cell node 10 has cell nodes 2, 3, 4 and 5 as its neighbors in the east direction. Cell nodes 2, 3, 4 and 5 could point their respective west pointers to cell node 10, but there is no way cell node 10 can point its only east pointer to the four neighbor cells simultaneously. For this reason, a transition node has been created in order to handle the communication between these nodes. Thus, the east pointer of cell node 10 is pointed to the transition node, whose four quadruple connectors points each to each of the four cell nodes 2, 3, 4 and 5. The single connector of the transition node is pointed to cell node 10 and the west pointer of each of the nodes 2, 3, 4 and 5 is pointed to the transition node.

2.2 Implementation

The implementation chosen in this paper is based on the object-oriented programming paradigm. The class *Cell* represents the nodes of the graph, irrespective of their types. Since there are two different types of nodes, two inherited classes were created: class *CellNode* represents the cell nodes and class *TransitionNode* represents the transition nodes. See Figure 3.

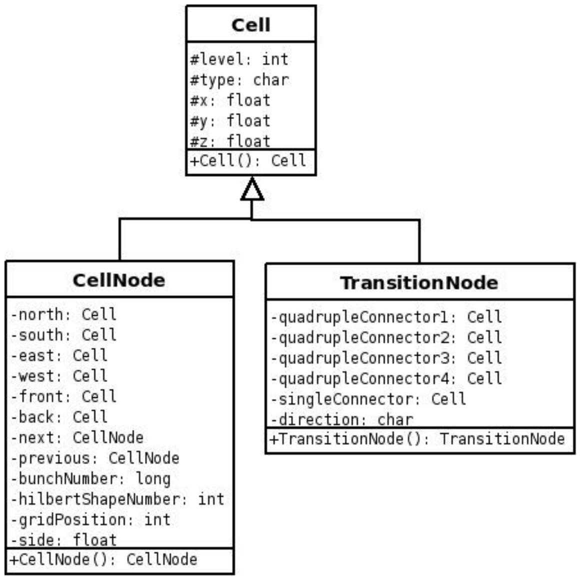


Fig. 3. Hierarchy of classes Cell, CellNode and TransitionNode.

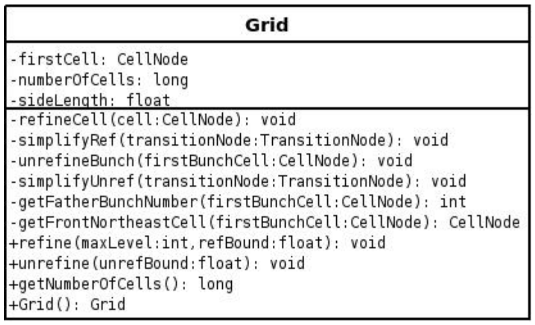
The ALG graph itself is encapsulated by the class *Grid*. The refinement and derefinement methods, as well as other methods of internal use for the maintenance of the data structure, are implemented as members of this class (see Figure 4).

When an object of type *Grid* is instantiated, the constructor generates an initial graph as the one of Figure 1. The steps necessary for the creation of this initial graph are described by the algorithm 1.

The remaining members of class *Grid*, which perform the functions of refinement, derefinement and ordering of the cell nodes via the Hilbert curve, will be discussed in the next sections.

3. REFINEMENT

The operation of *refining* a cell is defined as dividing it into eight equal cubic cells. When a cell is refined, the cell node which represents it in the graph structure is replaced by a structure consisting of eight cell nodes and six transition nodes,



```
Grid
-firstCell: CellNode
-numberOfCells: long
-sideLength: float
-refineCell(cell:CellNode): void
-simplifyRef(transitionNode:TransitionNode): void
-unrefineBunch(firstBunchCell:CellNode): void
-simplifyUnref(transitionNode:TransitionNode): void
-getFatherBunchNumber(firstBunchCell:CellNode): int
-getFrontNortheastCell(firstBunchCell:CellNode): CellNode
+refine(maxLevel:int,refBound:float): void
+unrefine(unrefBound:float): void
+getNumberOfCells(): long
+Grid(): Grid
```

Fig. 4. Class Grid.

Algorithm 1 Steps followed by the constructor of class Grid in order to create the initial graph.

```
1:  Create eight cell nodes.
2:  Create six transition nodes.
3:  $sideLength \leftarrow 1$
4:
5:  for each cell node do
6:      Compute and set current node's coordinates.
7:      Points each current node's directional pointers to appropriate cell.
8:      Set current node's previous and next pointers according to Hilbert curve's ordering.
9:      Set current node's level to one.
10:     Set current node's side to $sideLength/2$.
11:     Initialize other attributes pertinent to the current application.
12: end for
13:
14: for each transition node do
15:     Compute and set current node's coordinates.
16:     Point each current node's quadruple connector to appropriate cell node.
17:     Set current node's single connector to null.
18:     Set current node's level to one.
19: end for
20:
21: $numberOfCells \leftarrow 8$.
22: Set pointer $firstCell$ to the first cell of the Hilbert Curve list.
```

which is similar to the initial graph. In order to exemplify the refinement process, consider the refinement of cell number 0 in the mesh of Figure 5(a).
After refinement, cell 0 will be divided into eight smaller cells, half the length of the original one, with centers

$$\left(\frac{7}{8},\frac{7}{8},\frac{7}{8}\right),\left(\frac{7}{8},\frac{7}{8},\frac{5}{8}\right),\left(\frac{7}{8},\frac{5}{8},\frac{5}{8}\right),\left(\frac{7}{8},\frac{5}{8},\frac{7}{8}\right),$$

$$\left(\frac{5}{8},\frac{7}{8},\frac{7}{8}\right),\left(\frac{5}{8},\frac{7}{8},\frac{5}{8}\right),\left(\frac{5}{8},\frac{5}{8},\frac{5}{8}\right),\left(\frac{5}{8},\frac{5}{8},\frac{7}{8}\right),$$

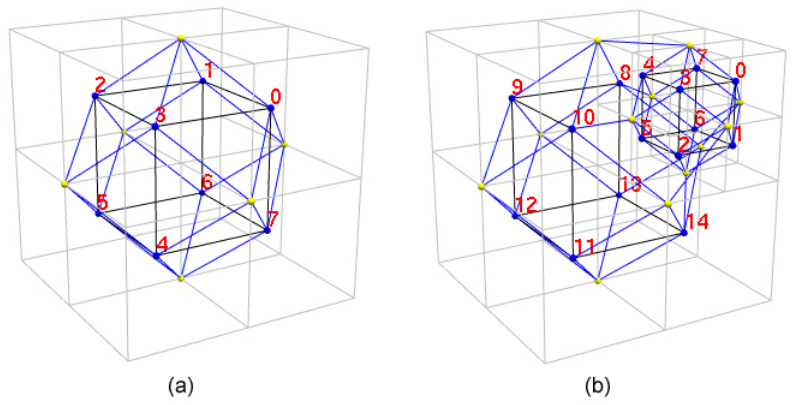


Fig. 5. Refinement of one cell.

resulting in the mesh of Figure 5(b).

More generally, when a cell centered at (a, b, c) and length L is refined, it is replaced by eight cells with length $L/2$ and centers

$$\left(a-\frac{L}{4}, b-\frac{L}{4}, c-\frac{L}{4}\right), \left(a+\frac{L}{4}, b-\frac{L}{4}, c-\frac{L}{4}\right), \left(a-\frac{L}{4}, b+\frac{L}{4}, c-\frac{L}{4}\right),$$

$$\left(a-\frac{L}{4}, b-\frac{L}{4}, c+\frac{L}{4}\right), \left(a+\frac{L}{4}, b+\frac{L}{4}, c-\frac{L}{4}\right), \left(a+\frac{L}{4}, b-\frac{L}{4}, c+\frac{L}{4}\right),$$

$$\left(a-\frac{L}{4}, b+\frac{L}{4}, c+\frac{L}{4}\right), \left(a+\frac{L}{4}, b+\frac{L}{4}, c+\frac{L}{4}\right).$$

The group of eight cells arising from a single original cell through refinement will be called a *bunch*. The cells of the new bunch are ordered by the modified Hilbert curve, as will be explained in Section 5. The creation and ordering of the new bunch of cells is merely a local event, which alters only the states of the neighboring cells of the original cell. These neighbor cells accordingly need to be notified of the local change in the graph structure.

A cell that was just refined receives as its refinement level the level of the original unrefined cell plus one. By convention, the cells of the initial mesh possess refinement levels equal to 1. In Figure 5(b), for instance, cells number 0 to 7 have refinement level 2, while the remaining cells possess refinement level 1. Transition nodes are also assigned refinement levels, equal to the refinement levels of the cells of the bunch in which they were created.

The length of a cubic cell L relates to its refinement level n through the simple relation

$$L = \frac{1}{2^n}. \tag{1}$$

3.1 Mesh refinement

Mesh refinement is done according to criteria laid out by the specific application. In the typical setting of adaptive mesh refinement (AMR), when the variation of some physical variable in the cells within a region of the computational domain exceeds some preset value, called here the *refinement bound*, indicating that more precision might be needed in order to obtain more accurate results, the program decides that the cells belonging to that region must be refined. The program traverses the entire mesh in each global iteration of the computations, verifying if any cell satisfies the refinement condition defined by the application. When a visited cell satisfies the condition, it is refined, originating a modified mesh and its associated graph for the next global iteration of the program.

Algorithm 2 describes function *refine()*, which is responsible for the mesh refinement. It receives as parameters the *minimum refinement level* desired for the mesh's cells, which is related to the minimum mesh size initially expected necessary in order to get a reasonably meaningful approximation to the solution of the problem being studied, and the refinement bound discussed above.

Algorithm 2 Mesh refinement

```
1:  procedure REFINE(minLevel, refinementBound)
2:      $CellNode\ currentCell$
3:      $CellNode\ auxiliar$
4:      $continueRefining \leftarrow true$
5:      while $continueRefining$ do
6:          $currentCell \leftarrow firstCell$
7:          $continueRefining \leftarrow false$
8:          while $currentCell$ is not $null$ do
9:              if $currentCell.level < minLevel$ and the local spatial variation of some
                property in the cell exceeds the refinement bound then
10:                 $auxiliar \leftarrow currentCell$
11:                 $currentCell \leftarrow currentCell.next$
12:                 $refineCell(auxiliar)$
13:                 $numberOfCells \leftarrow numberOfCells + 7$
14:                 $continueRefining \leftarrow true$
15:             else
16:                 $currentCell \leftarrow currentCell.next$
17:             end if
18:         end while
19:     end while
20: end procedure
```

Notice in Algorithm 2 that the attribute that stores the number of cell nodes increases by 7 at each cell refinement and not by 8. This happens because the cell node which originated the new bunch is not stored as it would be in a typical tree structure, but is completely replaced by the new bunch. Moreover, in order to augment the performance, the cell node is not actually destroyed upon refinement but is transformed into part of the new bunch (specifically, it becomes the front northeast cell node, as will be explained in the following). Thus, only 7 new cell nodes are created when a cell is refined.

3.2 Cell refinement

The refinement of one cell is done by the function *refineCell()*. The process occurs in seven steps as described below. Each step will be presented afterwards in a detailed algorithm.

(1) Transformation of the cell to be refined into the front northeast cell of the new bunch.
(2) Creation of the remaining 7 cell nodes of the new bunch.
(3) Creation of 6 transition nodes.
(4) Linking of the cell and transition nodes of the new bunch.
(5) Linking of the neighboring cells of the original cell to the transition nodes of the new bunch.
(6) Ordering of the cells of the new bunch using the modified Hilbert curve.
(7) Elimination of unnecessary transition nodes.

Step 1. Creation of the front northeast cell of new bunch. In step 1, the cell to be refined becomes the front northeast cell of the new bunch. Its attributes, such as geometric coordinates, refinement level and others which depend on the problem being considered, are accordingly updated at this stage, as described in Algorithm 3.

Algorithm 3 Step 1 of 7

```
1: procedure REFINECELL(cellNode)
2:     $fatherBunchNumber \leftarrow cellNode.bunchNumber$
3:     $CellNode\ \ frontNE \leftarrow cellNode$
4:     $frontNE.level \leftarrow cellNode.level + 1$
5:     $frontNE.side \leftarrow cellNode.side/2$
6:     $frontNE.x \leftarrow cellNode.x + cellNode.side/4$
7:     $frontNE.y \leftarrow cellNode.y + cellNode.side/4$
8:     $frontNE.z \leftarrow cellNode.z + cellNode.side/4$
9:     $frontNE.bunchNumber \leftarrow fatherBunchNumber * 10 + 1$
```

The meaning of the attributes *bunchNumber* and *fatherBunchNumber* will be explained in the next step.

Step 2. Creation of the remaining cells of the new bunch. In this step, the remaining seven cells of the bunch with all their attributes are created, as described in Algorithm 4.

In the derefinement process, those cells which originated from the same cell (i.e., cells that belong to the same bunch) must again become one cell, with some of the attributes of the original cell being restored. In order to easily identify the original cell, even after undergoing several steps of refinement, it is necessary that each cell stores information regarding the cell which originated it. This identification process is carried out through specific modifications of the identifier *bunchNumber* of each

Algorithm 4 Step 2 of 7

```
10: $CellNode\ backNE \leftarrow$ **new** $CellNode$          ▷ Back northeast cell.
11: $CellNode\ backNW \leftarrow$ **new** $CellNode$          ▷ Back northwest cell.
12: $CellNode\ frontNW \leftarrow$ **new** $CellNode$         ▷ Front northwest cell.
13: $CellNode\ frontSW \leftarrow$ **new** $CellNode$         ▷ Front southwest cell.
14: $CellNode\ backSW \leftarrow$ **new** $CellNode$          ▷ Back southwest cell.
15: $CellNode\ backSE \leftarrow$ **new** $CellNode$          ▷ Back southeast cell.
16: $CellNode\ frontSW \leftarrow$ **new** $CellNode$         ▷ Front southwest cell.
17:
18: for each new cell do
19:     sets current cell's level to $frontNE.level$
20:     sets current cell's side length to $frontNE.side$
21:     computes and sets current cell's coordinates
22:     computes and sets other attributes of current cell
23: end for
24:
25: $backNE.bunchNumber \leftarrow fatherBunchNumber * 10 + 2$
26: $backNW.bunchNumber \leftarrow fatherBunchNumber * 10 + 3$
27: $frontNW.bunchNumber \leftarrow fatherBunchNumber * 10 + 4$
28: $frontSW.bunchNumber \leftarrow fatherBunchNumber * 10 + 5$
29: $backSW.bunchNumber \leftarrow fatherBunchNumber * 10 + 6$
30: $backSE.bunchNumber \leftarrow fatherBunchNumber * 10 + 7$
31: $frontSW.bunchNumber \leftarrow fatherBunchNumber * 10 + 8$
```

cell at each refinement step. This attribute identifies each cell uniquely and stores information about its origin.

At each refinement step, the attribute *bunchNumber* of each cell of the new bunch is initially configured to be the *bunchNumber* of the cell being refined multiplied by 10, which corresponds to shifting one unit to the left the numerals of *bunchNumber* of the original cell. Since this identifier must be unique, a different numeral between 1 and 8 is then added to the result for each cell of the bunch.

Therefore, in order to identify the *bunchNumber* of the originating cell of any cell node of the graph, it suffices to divide the *bunchNumber* of the cell node by 10 and discard the remainder. Any cell node which belongs to the same bunch will produce the same number. Thus, the derefinement procedure will easily be able to identify when two cells belong to the same bunch.

Step 3. Creation of transition nodes of the new bunch. In order to complete the set of nodes of the new bunch, six transition nodes must be created. They are responsible for linking the bunch to the neighboring cells of the cell just refined, that is, link the new bunch to the graph. This stage is described by Algorithm 5.

At line 42 of Algorithm 5 the single connector of the transition node is pointed to the neighbor cell of *cellNode* (the refined cell) which is in the same direction as the transition node itself. For instance, north transition node points its single connector to the neighbor cell of *cellNode* which is pointed at by the pointer of *cellNode* in the direction *north*. The same happens to the remaining nodes, so that in the end of the loop all transition nodes are appropriately connected to the neighboring cells of the refined cell. Although transition nodes need not be assigned geometrical coordinates (since they do not exist physically in the computational domain), they are assigned coordinates for the purpose of drawing the graph if one so wishes.

In order to complete the two-directional link between nodes, there remains to point the node which was pointed to by the transition node to the transition node itself. In the example of the previous paragraph, if the node pointed to is a cell node, its pointer of direction *south* must be pointed to transition node *north*. If the node is a transition node, one of its quadruple connectors or its single connector must be pointed to transition node *north*. The verification of node type and the determination of the type of connector is done at step 5.

Step 4. Linking of cell and transition nodes. After the creation and initialization of cell and transition nodes of the new bunch, these must be linked, forming a structure as shown in Figure 6. These linkings are done by Algorithm 6.

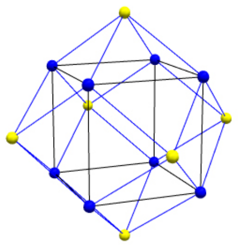


Fig. 6. Linking between cell and transition nodes.

The first loop of Algorithm 6 configures the directional pointers of each cell node in the bunch. For instance, pointer *north* of front northeast cell is pointed to the north transition node, while pointer *south* is pointed to the front southeast cell node. At the end of the loop, all directional pointers of all cell nodes will be appropriately configured.

The second loop configures the quadruple connectors of the transition nodes. A scheme is defined to determine to which cell each quadruple connector must

Algorithm 5 Step 3 of 7

```
32:   $TransitionNode\ northTN$ ← **new** $TransitionNode$   ▷ North transition node.
33:   $TransitionNode\ southTN$ ← **new** $TransitionNode$   ▷ South transition node.
34:   $TransitionNode\ eastTN$ ← **new** $TransitionNode$    ▷ East transition node.
35:   $TransitionNode\ westTN$ ← **new** $TransitionNode$    ▷ West transition node.
36:   $TransitionNode\ frontTN$ ← **new** $TransitionNode$   ▷ Front transition node.
37:   $TransitionNode\ backTN$ ← **new** $TransitionNode$    ▷ Back transition node.
38:
39:   for each new transition node do
40:       sets current node's level to $frontNE.level$
41:       computes and sets current node's coordinates
42:       points current node's single connector to appropriate cellNode's neighbor
43:   end for
```

Algorithm 6 Step 4 of 7

```
44:    for each cell node do
45:        updates each current cell's direction pointer
46:    end for
47:
48:    for each transition node do
49:        updates each current node's quadruple connector
50:    end for
```

point. This must be done in a very precise way, as this choice will interfere in the simplification procedure. Figure 7 shows the adopted scheme.

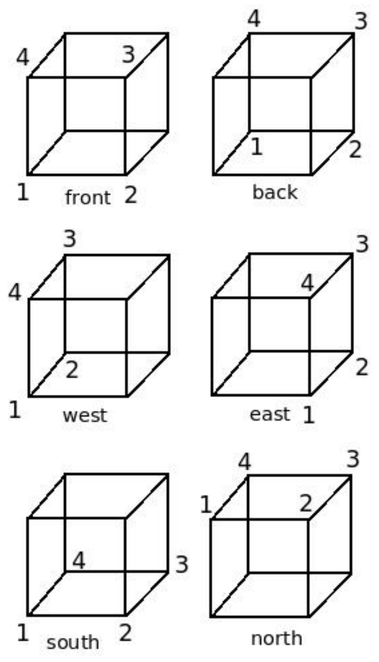


Fig. 7. Configuration scheme for quadruple connectors of the transition nodes.

For instance, as seen in the second cube in the second column of Figure 7, quadruple connectors 1, 2, 3 and 4 of transition node (*east*) are respectively pointed to node cells front southeast, back southeast, back northeast and front northeast.

Step 5. Linking of the graph to the new bunch. In step 3, after being created, the transition nodes of the new bunch were pointed to the neighboring cells of the just refined *cellNode* thereby linking it to the graph of the mesh. Reciprocally, the graph itself must be connected to the new bunch, since all links in the graph are bidirectional; at this point in the algorithm the neighboring nodes of the refined cell still point to *cellNode*, which no longer exists. This update is done by Algorithm 7, where the neighboring nodes to the refined cell are linked to the six transition nodes of the bunch.

The neighboring nodes of the refined cell *cellNode* can be either cell nodes or transition nodes. When the neighbor node is a cell node, the directional pointer of that node must be pointed to the transition node whose direction is opposite to that of the transition node. For example, assume that the north transition node has a cell node as a neighbor, that is, its single connector points to a cell node. This neighboring cell node must have its directional pointer *south* pointed to the north transition node. In Algorithm 7, this task is executed in line 66, where method *oppositeDirection()* returns the directional pointer whose direction is opposite to that received as a parameter.

In case the neighboring node of the transition node of the bunch is also a transi-

Algorithm 7 Step 5 of 7

```
51:     $char\ direction$
52:     $CellNode\ neighborCN$
53:     $TransitionNode\ neighborTN$
54:     $TransitionNode\ transitionNode[6]$
55:     $transitionNode[0] \leftarrow northTN$
56:     $transitionNode[1] \leftarrow southTN$
57:     $transitionNode[2] \leftarrow eastTN$
58:     $transitionNode[3] \leftarrow westTN$
59:     $transitionNode[4] \leftarrow frontTN$
60:     $transitionNode[5] \leftarrow backTN$
61:
62:     for i $\leftarrow$ 0 to 5 do
63:         if $transitionNode[i] \cdot singleConnector$ is a cell node then
64:             $neighborCN \leftarrow transitionNode[i].singleConnector$
65:             $direction \leftarrow transitionNode[i].direction$
66:             $neighborCN.oppositeDirection(direction) \leftarrow transitionNode[i]$
67:         else                    ▷ In this case the neighboring cell is a transition node
68:             $neighborTN \leftarrow transitionNode[i].singleConnector$
69:             if $neighborTN.singleConnector == cellNode$ then
70:                 $neighborTN.singleConnector \leftarrow transitionNode[i]$
71:
72:             else if $neighborTN.quadrupleConnector1 == cellNode$ then
73:                 $neighborTN.quadrupleConnector1 \leftarrow transitionNode[i]$
74:
75:             else if $neighborTN.quadrupleConnector2 == cellNode$ then
76:                 $neighborTN.quadrupleConnector2 \leftarrow transitionNode[i]$
77:
78:             else if $neighborTN.quadrupleConnector3 == cellNode$ then
79:                 $neighborTN.quadrupleConnector3 \leftarrow transitionNode[i]$
80:
81:             else if $neighborTN.quadrupleConnector4 == cellNode$ then
82:                 $neighborTN.quadrupleConnector4 \leftarrow transitionNode[i]$
83:             else
84:                 $print('Error')$
85:             end if
86:         end if
87:     end for
```

tion node, then an analysis of its single connector and all its quadruple connectors must be done in order to determine which one points to *cellNode*, which is the one who must be updated and pointed to the transition node of the bunch. This analysis and update are done by the piece of code between lines 67 and 86.

Step 6. Ordering of the cell nodes of the graph using the modified Hilbert curve. The modified Hilbert curve is a space filling curve which is capable of passing through all the points of a three-dimensional mesh irrespective of how irregular this mesh might be. Its implementation is done by means of a double chain list where each cell node possesses two additional pointers: *next* and *previous*. Any cell of the mesh can be reached traversing the chain list defined by the Hilbert curve to the point it occupies.

When a cell is refined, the 8 new cells of the bunch must be ordered and inserted in the Hilbert curve, in order to be accessible later. They are ordered in a small list, according to the relative position of the refined cell in the Hilbert curve of the graph (before refinement), which is later inserted in the main list. This preordering in a sublist guarantees that the cells of the same bunch will always be near each other in the ordering of the curve, causing only a local modification in the curve. Therefore, inserting the new cells in the main curve consists only in inserting an already ordered sublist in the position of the cell just refined. The manner in which this is done is detailed in Section 5. However, the implementation is done at this step of the refinement, where the references for the pointers *next* and *previous* of each cell of the bunch are defined. Algorithm 8 shows how this is executed.

Algorithm 8 Step 6 of 7

```
88:   for each cell node do
89:       sets current cell's next pointer according to modified Hilbert curve algorithm
90:       sets current cell's previous pointer according to modified Hilbert curve algorithm
91:   end for
```

Step 7. Removal of unnecessary transition nodes of the graph. The function of the transition nodes in the ALG data structure is to enable the communication between nodes of different levels of refinement. In the refinement process, transition nodes are created to connect the cells of the new bunch with the neighboring cells of the refined cell. However, the neighborhood of the refined cell may contain cells which have the same level of refinement of the cells of the new bunch. In this situation a transition node is dispensable, since the cells can be directly linked.

For instance, in Figure 8, we have two neighboring bunches whose cell nodes possess the same refinement levels and which are connected by means of two transition nodes. However, they can be directly connected as shown in Figure 9.

In order to eliminate the storage problem associated with keeping unnecessary transition nodes, the last step in the refinement process contains a *simplification* procedure that deletes needless transition nodes in the new bunch and its immediate neighborhood, as well as makes the direct link between nodes with the same refinement level. This procedure is detailed in the next subsection.

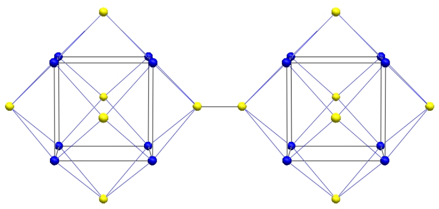


Fig. 8. Graph before simplification showing two unnecessary transition nodes.

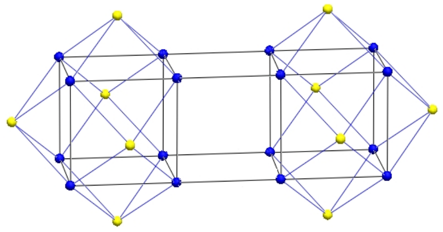


Fig. 9. Simplified graph with transition nodes eliminated and cell nodes directly linked.

Algorithm 9 Step 7 of 7

```
92:     for i ← 0 to 5 do
93:         $simplifyRef(transitionNode[i])$
94:     end for
95: end procedure
```

3.3 Simplification routine

The simplification routine is executed by the code described in Algorithm 10. In order for a simplification to occur, two conditions must be satisfied. The first one is that the single connector of a transition node received as a parameter in function *simplifyRef()* must point to a transition node. The second condition is that they must possess the same refinement level. In Algorithm 10 these conditions are verified in lines 2 and 3.

When both conditions are obeyed, the graph must be simplified. Before removing the unnecessary transition nodes, the references of their quadruple connectors must be stored, since they are the references for the cell nodes which will be directly linked.

In Algorithm 10, vector *quadCon* stores references for the quadruple connectors of the transition node (*transitionNode*) received as a parameter; vector *nQuadCon* stores references for the quadruple connectors of the neighbor transition node (*neighborTN*) which is pointed to by the single connector of *transitionNode*.

Thereafter, the cells of the bunch which are referenced by the quadruple connectors of *transitionNode* are pointed to those referenced by the quadruple connectors of *neighborTN*. This is done by the loop between lines 20 and 22. The directional

Algorithm 10 Simplification - Refinement

```
procedure SIMPLIFYREF(transitionNode)
    if $transitionNode.singleConnector.type == transitionNode.type$ then
        if $transitionNode.singleConnector.level == transitionNode.level$ then

            $TransitionNode\ neighborTN \leftarrow transitionNode.singleConnector$
            $char\ dir \leftarrow neighTN.direction$
            $CellNode\ quadCon[4]$        ▷ Stores the node's quadruple connectors
            $Cell\ nQuadCon[4]$        ▷ Stores the neighbor node's quadruple connectors

            $quadCon[0] \leftarrow transitionNode.quadrupleConnector1$
            $quadCon[1] \leftarrow transitionNode.quadrupleConnector2$
            $quadCon[2] \leftarrow transitionNode.quadrupleConnector3$
            $quadCon[3] \leftarrow transitionNode.quadrupleConnector4$

            $nQuadCon[0] \leftarrow neighborTN.quadrupleConnector1$
            $nQuadCon[1] \leftarrow neighborTN.quadrupleConnector2$
            $nQuadCon[2] \leftarrow neighborTN.quadrupleConnector3$
            $nQuadCon[3] \leftarrow neighborTN.quadrupleConnector4$

            for $i \leftarrow 0$ to 3 do
                $quadCon[i].oppositeDirection(dir) \leftarrow nQuadCon[i]$
            end for
            $dir \leftarrow transitionNode.direction$
            for $i \leftarrow 0$ to 3 do
                if $nQuadCon[i]$ is a cell node then
                    $nQuadCon[i].oppositeDirection(dir) \leftarrow quadCon[i]$
                else
                    if $neighborTN == nQuadCon[i].singleConnector$ then
                        $nQuadCon[i].singleConnector \leftarrow quadCon[i]$
                    else if $neighborTN == nQuadCon[i].quadrupleConnector1$ then
                        $nQuadCon[i].quadrupleConnector1 \leftarrow quadCon[i]$
                    else if $neighborTN == nQuadCon[i].quadrupleConnector2$ then
                        $nQuadCon[i].quadrupleConnector2 \leftarrow quadCon[i]$
                    else if $neighborTN == nQuadCon[i].quadrupleConnector3$ then
                        $nQuadCon[i].quadrupleConnector3 \leftarrow quadCon[i]$
                    else if $neighborTN == nQuadCon[i].quadrupleConnector4$ then
                        $nQuadCon[i].quadrupleConnector4 \leftarrow quadCon[i]$
                    else
                        $print('Error')$
                    end if
                end if
            end for

            delete $transitionNode$
            delete $neighborTN$
        end if
    end if
end procedure
```

pointer of each one of these nodes that must be reconfigured is the one whose direction is opposite of that of *neighborTN*. For instance, assume in Figure 8 that the west transition node of the right-hand bunch is passed as a parameter to the simplification routine. Then the nodes which are referenced by its quadruple connectors will point their *west* directional pointers to the nodes referenced by the quadruples connectors of the east transition node of the left-hand side bunch. Here the scheme depicted in Figure 7 guarantees that the connections are correctly done.

Next the cells of the bunch which are referenced by the quadruple connectors of *neighborTN* are pointed to those referenced by the quadruple connectors of *transitionNode*. These nodes can be either cell nodes or transition nodes. If they are cell nodes, the directional pointer that must be reconfigured is that with direction opposite to the one of *transitionNode*. This is done in line 26 of Algorithm 10. If they are transition nodes, one must first find out which type of connector of these nodes points to *neighborTN*. This is done by the piece of code between lines 27 and 45.

After all neighbor nodes with the same refinement levels are connected, the transition nodes *transitionNode* and *neighborTN* are removed from the graph.

4. DEREFINEMENT

Derefinement means the fusion of eight cells with the same level of refinement n and belonging to the same bunch back into a unique cell of refinement level $n-1$ (Figures 10 and 11).

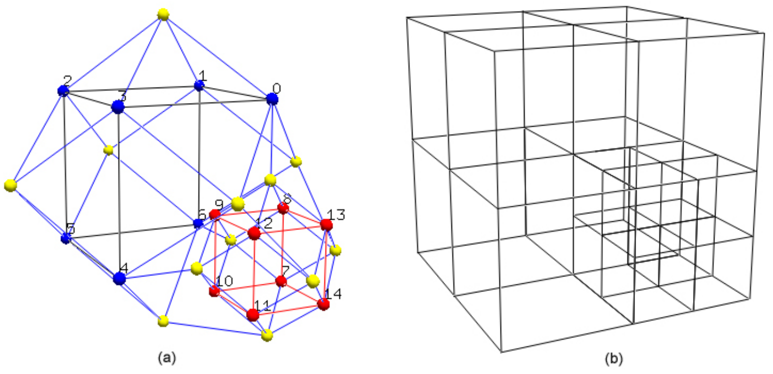


Fig. 10. Graph and mesh before derefinement.

The decision to derefine or not a bunch is taken according to criteria established by the application. Generally, one derefines a bunch when the variation in some physical variable within the cells in a certain neighborhood is below some preset threshold. When this happens, the number of cells in the region becomes greater than that necessary in order to accomplish an accurate approximation of the exact solution in that region, causing wastage of memory and processing power.

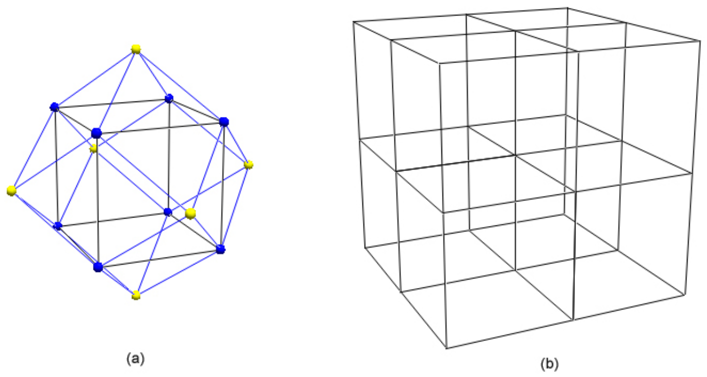


Fig. 11. Graph and mesh after derefinement.

4.1 Mesh derefinement

The derefinement of the mesh is done by the method *derefine()*. Whenever it is called upon, a research is done on all cells of the mesh to determine which ones are qualified for derefinement according to the criteria explained above.

Before a bunch is actually derefined, the cells initially qualified for derefinement must pass some tests. First, all cells must belong to the same bunch, otherwise it will not be allowed that the 8 adjacent cells which define a cube in the mesh be derefined, even if they satisfy the derefinement condition adopted by the application. Second, all cells in the bunch must satisfy the derefinement condition. Algorithm 11 presented below resumes what is done by method *derefine()*. It receives as a parameter the maximum variation accepted by one or more physical variables upon which the derefinement is based.

Notice that in Line 5 the algorithm verifies if the current cell and its seven adjacent cells belong to the same bunch. This happens when they have the same refinement level and the division without remainder of the attribute *bunchNumber* of each cell wields the same result.

Function *derefineBunch()* called in Line 11 receives as a parameter the first cell of the bunch to be refined. Its task is to derefine the bunch, as will be discussed next.

4.2 Bunch derefinement

The bunch derefinement procedure has as a parameter the first cell of the bunch to be refined, as the mesh derefinement algorithm encounters when it follows the ordering established by the Hilbert curve. In order to obtain the remaining cells of the bunch, it just follows the order given by the Hilbert curve until the seventh next cell, because the Hilbert curve is built in such a way as to ensure that all cells belonging to the same bunch follow in a sequence one after the other in the chain list defined by the curve. Therefore, is suffices to access the first cell of the list in order to obtain access to the remaining cells of the bunch.

The entire process is divided below in six steps, each step discussed in detail in

Algorithm 11 Mesh derefinement

```
procedure DEREFINE(maxPrecision)
    $CellNode\ \ currentCell \leftarrow firstCell$
    $CellNode\ \ cellAfterBunch \leftarrow currentCell$
    while $currentCell$ is not $null$ do
        if $currentCell$ and its seven adjacent cells belong to the same bunch then
            if all cells of current bunch have the specified properties $> maxPrecision$ then
                $cellAfterBunch \leftarrow currentCell$
                for i $\leftarrow$ 0 to 8 do
                    $cellAfterBunch \leftarrow cellAfterBunch.next$
                end for
                $derefineBunch(currentCell)$
            end if
        end if
        $currentCell \leftarrow cellAfterBunch$
        if $cellAfterBunch$ is not $null$ then
            $cellAfterBunch \leftarrow cellAfterBunch.next$
        end if
    end while
end procedure
```

the next paragraphs.

(1) Creation of the cell node which will replace the bunch.
(2) Creation of six transition nodes around the replacer cell node.
(3) Linking between the cell and transition nodes just created.
(4) Linking of the new transition nodes to the neighboring nodes of the bunch being derefined.
(5) Elimination of all nodes of the bunch to be derefined.
(6) Removal of unnecessary transition nodes in the neighborhood of the derefined bunch.

Step 1. Creation of the cell node which replaces the bunch. The derefinement procedure produces a cell with characteristics which are similar to the cell which originated the bunch which is being derefined. Its attributes are computed from the corresponding attributes of the cells in the bunch. In the end, the group of eight cells which form the bunch is replaced by the recently created cell.

In order to avoid the costs of memory allocation and management associated with the creation of a new cell, one uses a preexistent cell of the bunch to replace it. In this implementation, the front northeast cell was chosen. Algorithm 12 (Lines 11 through 19) shows how this cell recovers some of the geometric and topological attributes of the original cell that gave origin to the bunch. Lines 12 through 14 reconfigure the geometric coordinates. Line 15 computes the identifier of the replacer cell, restoring it to the value of the cell which originated the bunch: this identifier was built in such a way as to guarantee that all cells belonging to the same bunch produce the same value when their identifier is divided without remainder by 10 (see Step 2 in Section 3). Line 16 computes the value of the basic form of the

Hilbert curve that will be attributed to the replacer cell; this value is read by the refinement procedure, in case it is decided in another iteration of the program that this cell must be refined again, in order to tell how the cells of the resulting bunch will be ordered by the Hilbert curve (see Section 5). This information is passed on by the cells of the present bunch themselves, which give the replacer cell the precise value so that in the future the cells of the newly refined bunch should be ordered in the same way. This procedure is fundamental for the correct functioning of the Hilbert curve. A call to the function *getFatherHilbertShape()* returns the number of the Hilbert curve shape that was used in the refinement routine for the ordering of the cells in the current bunch. Lines 17 and 18 update the refinement level and cell length, respectively. Other variables which depend on the problem being considered should be updated in this stage of the derefinement as well.

Algorithm 12 Step 1 of 6

```
 1: procedure DEREFINEBUNCH(firstBunchCell)
 2:     $CellNode\ \ frontNE \leftarrow getFrontNortheastCell(firstBunchCell)$
 3:     $CellNode\ \ frontNW \leftarrow frontNE.west$
 4:     $CellNode\ \ frontSE \leftarrow frontNE.south$
 5:     $CellNode\ \ frontSW \leftarrow frontNW.south$
 6:     $CellNode\ \ backNE \leftarrow frontNE.back$
 7:     $CellNode\ \ backNW \leftarrow backNE.west$
 8:     $CellNode\ \ backSE \leftarrow backNE.south$
 9:     $CellNode\ \ backSW \leftarrow backNW.south$
10:
11:     $CellNode\ \ replacerCell \leftarrow frontNE$
12:     $replacerCell.x \leftarrow (replacerCell.x + replacerCell.back.x)/2$
13:     $replacerCell.y \leftarrow (replacerCell.y + replacerCell.west.y)/2$
14:     $replacerCell.z \leftarrow (replacerCell.z + replacerCell.south.z)/2$
15:     $replacerCell.bunchNumber \leftarrow firstBunchCell.bunchNumber/10$
16:     $replacerCell.HilbertShapeNumber \leftarrow getFatherHilbertShape(firstBunchCell)$
17:     $replacerCell.level \leftarrow firstBunchCell.level - 1$
18:     $replacerCell.side \leftarrow cellNode.side * 2$
19:     updates other attributes.
20:
21:     $CellNode\ \ cellBeforeBunch \leftarrow firstBunchCell.previous$
22:     $CellNode\ \ cellAfterBunch \leftarrow firstBunchCell$
23:     for i $\leftarrow$ 0 to 8 do
24:         $cellAfterBunch \leftarrow cellAfterBunch.next$
25:     end for
26:     $replacerCell.previous \leftarrow cellBeforeBunch$
27:     $replacerCell.next \leftarrow cellAfterBunch$
28:     if $replacerCell.previous$ is not null then
29:         $replacerCell.previous.next \leftarrow replacerCell$
30:     end if
31:     if $replacerCell.next$ is not null then
32:         $replacerCell.next.previous \leftarrow replacerCell$
33:     end if
```

It is necessary to store references for all cell nodes in the bunch. This is done in Lines 2 through 9 in Algorithm 12. In order to find out the front northeast cell of the bunch, function *getFrontNortheastCell()* is called. Its determination is done analyzing the geometric coordinates of all cells in the bunch.

The code from Lines 21 through 33 reconfigures the pointers *previous* and *next* from the chain list defined by the Hilbert curve. Since the cell nodes from the bunch will be removed from the graph in the last stages of derefinement, it is necessary that they be removed also from this chain list. The removal action consists simply in substituting the replacer cell node in place of the sublist of the bunch cells.

Step 2. Creation of the six transition nodes of the replacer cell node. The replacer cell must have its directional pointers pointed to the neighboring nodes of the bunch which is being derefined. The refinement level of these nodes may be different from the replacer cell, making it necessary the existence of transition nodes for the communication of nodes having different refinement levels. Therefore a transition node is created for each direction. Notice that, if in some direction the cells nodes have the same refinement level, no transition node would be necessary in that direction. Nonetheless, transition nodes are initially created in every direction; those unneeded will be later removed in the simplification procedure (see Step 6). Although in principle one could design the derefinement procedure in such a way as not to create unnecessary transition nodes from the start, this would unreasonably complicate matters, making the algorithm difficult to understand and not gaining much in terms of performance, since a simplifying procedure would be needed anyway in order to get rid of unnecessary transition nodes present in the neighborhood of the replaced bunch due to its neighboring cells.

Figure 12 shows how the new transition nodes (white circles) are related to the replacer cell and the replaced bunch neighbors. The single connector of each node is pointed to the replacer cell, while the quadruple connectors point to the neighbors. Algorithm 13 shows how these nodes are created. Note in Line 42 that the refinement level of each transition node is configured to be one unit less than the refinement level of the bunch, that is, to match the refinement level of the replacer cell.

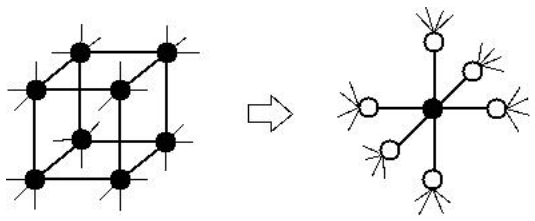


Fig. 12. Creation of transition nodes for the replacer cell.

Step 3. Linking between the replacer cell node and its transition nodes. After the creation of the transition nodes, it is necessary to link them to the replacer cell. This consists simply in pointing the single connector of each transition node to the replacer cell. Next, each directional pointer of the replacer cell must be pointed to the transition node located in the appropriate direction, i.e., north pointer should

Algorithm 13 Steps 2 and 3 of 6.

```
34:  $TransitionNode\ \ northTN \leftarrow$ **new** $TransitionNode$     ▷ North node.
35:  $TransitionNode\ \ southTN \leftarrow$ **new** $TransitionNode$     ▷ South node.
36:  $TransitionNode\ \ eastTN \leftarrow$ **new** $TransitionNode$      ▷ East node.
37:  $TransitionNode\ \ westTN \leftarrow$ **new** $TransitionNode$      ▷ West node.
38:  $TransitionNode\ \ frontTN \leftarrow$ **new** $TransitionNode$     ▷ Front node.
39:  $TransitionNode\ \ backTN \leftarrow$ **new** $TransitionNode$      ▷ Back node.
40:
41:  for each new transition node do
42:      sets current node's refinement level to $firstBunchCell.level - 1$
43:      computes and sets current node's coordinates
44:      points current node's single connector to $replacerCell$
45:  end for
46:
47:  $replacerCell.north \leftarrow northTN$
48:  $replacerCell.south \leftarrow southTN$
49:  $replacerCell.east \leftarrow eastTN$
50:  $replacerCell.west \leftarrow westTN$
51:  $replacerCell.front \leftarrow frontTN$
52:  $replacerCell.back \leftarrow backTN$
```

be directed to the north transition node and so on. In Algorithm 13, the first operation is executed in Line 44 while the second operation is executed in Lines 47 through 52.

Step 4. Linking between the replacer cell transition nodes and neighbor nodes of derefined bunch. In this stage, the quadruple connectors of the transition nodes are pointed to the neighboring nodes of the replaced bunch. The scheme represented in Figure 7 is again used here to ensure coherence. For instance, the quadruple connectors of the north transition node must be pointed to the nodes pointed to by the north directional pointers of each cell in the north face of the bunch. According to the scheme of Figure 7, the quadruple connectors 1, 2, 3 and 4 must be pointed to the north neighbor node of the cells front northwest, front northeast, back northeast and back northwest, respectively. These connections are done by Algorithm 14.

Step 5. Elimination of nodes of derefined bunch. At this point, the replacer cell is completely built and properly inserted in the graph. Therefore, the cells which made up the bunch are no longer needed and must be removed from the graph. The removal is straightforward and executed in the first lines of Algorithm 15. Notice that the front northeast cell is not removed, since it was chosen in the beginning to become the replacer cell.

Step 6. Removal of unnecessary transition nodes. As in the refinement process, after the derefinement of a bunch there may occur in the new graph transition nodes that connect cell nodes with the same level of refinement. These cells serve no purpose in the graph and must be eliminated. In order to do this, each recently created transition node goes through a simplification procedure, which identifies the unnecessary nodes and removes them. Moreover, the simplification procedure makes the direct link between cell nodes with the same refinement level and com-

Algorithm 14 Step 4 of 6

```
53:     $northTN.quadrupleConnector1 \leftarrow frontNW.north$
54:     $northTN.quadrupleConnector2 \leftarrow frontNE.north$
55:     $northTN.quadrupleConnector3 \leftarrow backNE.north$
56:     $northTN.quadrupleConnector4 \leftarrow backNW.north$
57:
58:     $southTN.quadrupleConnector1 \leftarrow frontSW.south$
59:     $southTN.quadrupleConnector2 \leftarrow frontSE.south$
60:     $southTN.quadrupleConnector3 \leftarrow backSE.south$
61:     $southTN.quadrupleConnector4 \leftarrow backSW.south$
62:
63:     $eastTN.quadrupleConnector1 \leftarrow frontNE.east$
64:     $eastTN.quadrupleConnector2 \leftarrow frontSE.east$
65:     $eastTN.quadrupleConnector3 \leftarrow backSE.east$
66:     $eastTN.quadrupleConnector4 \leftarrow backNE.east$
67:
68:     $westTN.quadrupleConnector1 \leftarrow frontNW.west$
69:     $westTN.quadrupleConnector2 \leftarrow frontSW.west$
70:     $westTN.quadrupleConnector3 \leftarrow backSW.west$
71:     $westTN.quadrupleConnector4 \leftarrow backNW.west$
72:
73:     $frontTN.quadrupleConnector1 \leftarrow frontSW.front$
74:     $frontTN.quadrupleConnector2 \leftarrow frontSE.front$
75:     $frontTN.quadrupleConnector3 \leftarrow frontNE.front$
76:     $frontTN.quadrupleConnector4 \leftarrow frontNW.front$
77:
78:     $backTN.quadrupleConnector1 \leftarrow backSW.back$
79:     $backTN.quadrupleConnector2 \leftarrow backSE.back$
80:     $backTN.quadrupleConnector3 \leftarrow backNE.back$
81:     $backTN.quadrupleConnector4 \leftarrow backNW.back$
```

pletes the connection of the neighboring nodes of the replaced bunch to the replacer cell (in Step 4, the quadruple connectors of each transition node were pointed to the neighboring nodes but these nodes did not point any of their directional nodes to the node which pointed to them). Thus, beside eliminating needless transition nodes, the simplification routine in the derefinement procedure also completes the connection of the replacer cell to the graph. The simplification routine for the derefinement procedure is explained in detail in the next section.

4.3 Simplification routine

When a bunch is derefined, six transition nodes are created around the replacer node. These nodes are connected to the replacer node and the neighboring nodes of the replaced bunch. It may happen that one of the neighboring cell nodes has the same refinement level as the replacer cell. In this case, the two cell nodes must be directly linked.

The connection between a replacer node and its neighbor can present several different configurations, depending on the refinement levels and types of its neighbors. Figure 13 shows the three different possible situations.

Algorithm 15 Step 5 of 6

```
    delete $frontNW$
    delete $frontSW$
    delete $frontSE$
    delete $backNE$
    delete $backNW$
    delete $backSE$
    delete $backSW$

    $simplifyDeref(northTN)$
    $simplifyDeref(southTN)$
    $simplifyDeref(eastTN)$
    $simplifyDeref(westTN)$
    $simplifyDeref(frontTN)$
    $simplifyDeref(backTN)$
end procedure
```

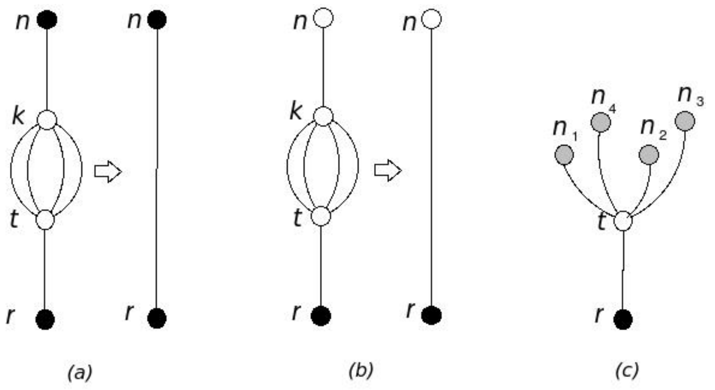


Fig. 13. Simplification through elimination of transition nodes.

In Figure 13, the nodes denoted by letters r, t and k are, respectively, the replacer cell node, a transition node created during derefinement, and the transition node which links to the neighbor node n of the original bunch.

The simplification procedure happens in the following manner:

(1) If node r has the same refinement level as node n, as it happens in cases *(a)* and *(b)*, r and n are directly linked. Next, nodes t and k are removed from the graph. Notice that the difference between cases *(a)* and *(b)* is that in the first r and n have the same type, while in the latter r is a cell node and n is a transition node.

(2) If the quadruple connectors of t point to different cells, then no simplification should be done, as in case *(c)*, since this means that n and the nodes n_i, $1 \leqslant i \leqslant 4$, have different refinement levels or are of different types.

Algorithm 16 executes the simplifying steps described above. In the third case, each node n_i must point to the transition node t. Notice that it is not necessary to point t to each node n_i, since this was already done in Step 4.

The code from Line 13 through 15 does the simplification for case *(a)*. The pointer of n (*neighborCN*) which points to k (*auxiliarTN*) must be modified to point to r (*replacerCell*). This pointer is that whose direction is opposite to the direction of transition node t (*transitionNode*). The elimination of nodes t and k occurs in Lines 32 and 33, respectively.

The simplification of the second case is executed by the code in Lines 16 through 27. As in case *(a)*, the pointer of n (now designated *neighborTN*) that points to k must be altered to point to r. Since n is a transition node and one cannot know a priori which of its five pointers points to k, an inspection must be obligatorily done on all pointers of n to find out which one points to k; once this is found, it is modified to point to r. After the linking, nodes t and k are removed from the graph.

In the third case, each node n_i (designated by the vector *quadCon*) must point to the transition node t. These nodes may be each either a cell node or a transition node. If it is a transition node, its single connector is pointed to t; if it is a cell node, the directional pointer whose direction is opposite to the direction of t is the one which is pointed to t. The distinction between these two cases occurs in Lines 40 through 43.

Algorithm 16 Simplification - Derefinement

```
procedure SIMPLIFYDEREF(transitionNode)
    $CellNode\ \ replacerCell \leftarrow transitionNode.singleConnector$
    $CellNode\ \ neighborCN \leftarrow null$
    $TransitionNode\ \ neighborTN \leftarrow null$
    $TransitionNode\ \ auxiliarTN \leftarrow null$
    $Cell\ \ quadCon[4] \leftarrow null$
    $char\ \ direction \leftarrow transitionNode.direction$

    if all transitionNode's quadruple connectors point to the same cell then
        $auxiliarTN \leftarrow transitionNode.quadrupleConnector1$
        $replacerCell.getNeighborCell(direction) \leftarrow auxiliarTN.singleConnector$
        $neighborType \leftarrow auxiliarTN.singleConnector.type$
        if the neighbor cell is a cell node then
            $neighborCN \leftarrow auxiliarTN.singleConnector$
            $neighborCN.oppositeDirection(direction) \leftarrow replacerCell$
        else
            $neighborTN \leftarrow auxiliarTN.singleConnector$
            if $neighborTN.singleConnector == auxiliarTN$ then
                $neighborTN.singleConnector \leftarrow replacerCell$
            else if $neighborTN.quadrupleConnector1 == auxiliarTN$ then
                $neighborTN.quadrupleConnector1 \leftarrow replacerCell$
            else if $neighborTN.quadrupleConnector2 == auxiliarTN$ then
                $neighborTN.quadrupleConnector2 \leftarrow replacerCell$
            else if $neighborTN.quadrupleConnector3 == auxiliarTN$ then
                $neighborTN.quadrupleConnector3 \leftarrow replacerCell$
            else if $neighborTN.quadrupleConnector4 == auxiliarTN$ then
                $neighborTN.quadrupleConnector4 \leftarrow replacerCell$
            else
                $print('Error')$
            end if
        end if
        delete $transitionNode$
        delete $auxiliarTN$
    else        ▷ In this case the neighbor cells are only linked, not simplified (3$^{rd}$ case)
        $quadCon[0] \leftarrow transitionNode.quadrupleConnector1$
        $quadCon[1] \leftarrow transitionNode.quadrupleConnector2$
        $quadCon[2] \leftarrow transitionNode.quadrupleConnector3$
        $quadCon[3] \leftarrow transitionNode.quadrupleConnector4$
        for $i \leftarrow 0$ to 3 do
            if $quadCon[i]$ is a transition node then
                $neighborTN \leftarrow quadCon[i]$
                $neighborTN.singleConnector \leftarrow transitionNode$
            else                                        ▷ the neighbor cell is a cell node
                $neighborCN \leftarrow quadCon[i]$
                $neighborCN.oppositeDirection(direction) \leftarrow transitionNode$
            end if
        end for
    end if
end procedure
```

5. THE MODIFIED HILBERT CURVE

The three-dimensional modified Hilbert curve is a space-filling curve which is used in ALG for its ability to provide an easy, efficient and predictable way of ordering the cells of the mesh independently of its irregularity and at the same time for having the property of making it possible to divide the mesh in relatively independent submeshes (see Figure 14). Its insensitivity to the irregularity of the mesh is mainly due to the fact that local changes in the mesh require only local changes in the curve. This fact also accounts for its ability to isolate different regions of the mesh. The subdivision of the mesh into independent regions through the Hilbert curve allows the potential use of parallel computing, since different processes can work simultaneously on different cell groupings.

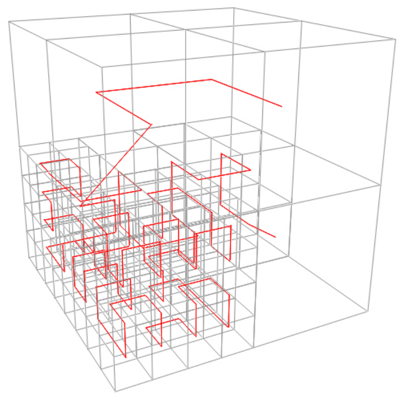


Fig. 14. Modified Hilbert curve on an irregular mesh.

5.1 Implementation

The modified Hilbert curve is implemented as a double chain list: every cell in the mesh, besides having six directional pointers, possesses also a pointer for the next cell and a pointer for the previous cell. Thus any cell in the mesh can be reached by traversing the list until it is located.

5.1.1 *The basic shapes and the Hilbert shape number.* The curve is built recursively by combining 12 different ways of ordering the eight cells of a bunch (see Figure 15). Each of these orderings is called a *basic shape* of the Hilbert curve and constitute the base cases for the recursion. Initially, one of the basic shapes is chosen for the ordering of the initial mesh. When one of the cells of this initial mesh is refined, the new bunch is also ordered according to one of the 12 basic shapes. After the ordering, the segment of curve in the new bunch is connected to the main curve forming a unique continuous curve.

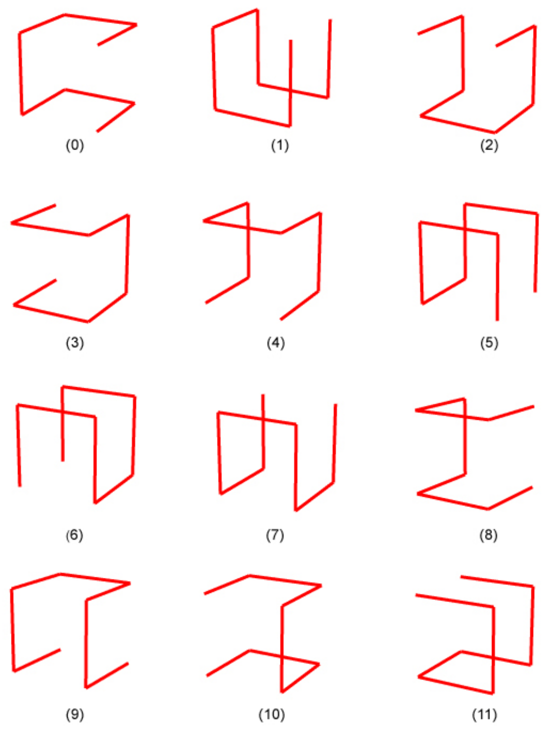


Fig. 15. Basic shapes of the Hilbert curve.

Each cell in the initial graph stores the number of the basic shape that will be used in the ordering of the cells in the new bunch in case it is refined, which is called the *Hilbert shape number*. When a refinement occurs, the newly created bunch is always ordered according to the basic shape informed by the refined cell. Each of the new cells of this bunch also stores the number of the basic shape which will be used in case the cell itself is also refined. This way, the modified Hilbert curve is recursively defined from the 12 basic shapes. This process does not require the mesh to be uniformly refined, differently from the traditional Hilbert curve which works only on regular meshes. The modified Hilbert curve has the advantage of supporting local changes in the mesh, when only one cell is refined or only one bunch is derefined.

As an example of this procedure, consider the refinement of cell number zero in Figure 16 (a). Upon refinement, it will be replaced by a bunch that will be ordered according to the Hilbert shape number stored in the original cell. For cell zero, this was set to be number 1. After the ordering of the bunch's cells using the basic shape 1, the main curve and the segment of curve in the new bunch are connected, resulting in the curve of Figure 16 (b).

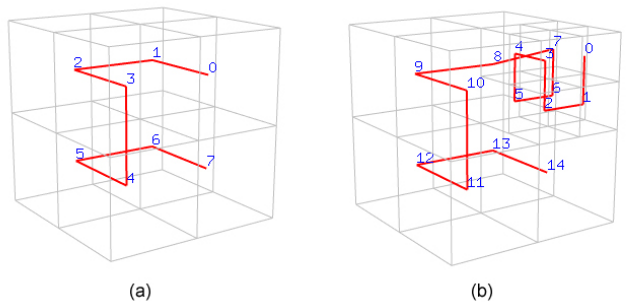


Fig. 16. Example of reordering of mesh cells using the modified Hilbert curve after the refinement of one cell.

In this work, the convention was made that the cells in the initial mesh will be always ordered using the basic shape number zero of Figure 15. Afterwards, the way a basic shape number is assigned to a cell in the mesh depends on the position of the cell in its bunch and the number of the basic shape that was used in ordering the bunch.

Table I shows how to define the Hilbert shape number for each cell in a bunch, given the number of the basic shape used to order the cells in the bunch to which the cell belongs and the position of the cell within the bunch. For instance, if the front southwest (F-SW) cell of a bunch ordered using basic shape 7 is refined, then the cells in the bunch that replaces this cell will be ordered using basic shape 11; correspondingly, these new cells will have their Hilbert shape number set also to 11.

Table I. Hilbert shape number assignment to mesh cells. Leftmost column refers to basic shape number previously used in ordering the bunch to which the cell belongs.

	F-NE	**F-SE**	**F-SW**	**F-NW**	**B-NE**	**B-SE**	**B-SW**	**B-NW**
0	1	5	3	2	2	4	4	2
1	0	2	2	6	6	7	7	8
2	1	9	10	11	0	0	10	10
3	7	9	9	7	0	0	6	11
4	7	5	6	7	0	0	10	10
5	4	0	11	4	9	8	11	9
6	4	1	10	4	9	1	3	9
7	4	1	11	4	3	11	1	8
8	7	9	9	7	1	5	10	10
9	8	8	3	3	2	5	6	2
10	8	8	6	11	2	4	4	2
11	10	10	5	2	3	3	5	7

The Hilbert shape number of each of the cells in the initial mesh is configured according to the first row in the table, corresponding to a Hilbert shape number equal to 0 for the bunch. Thus, the front northeast (F-NE) cell stores 1 in its

Hilbert shape number, the front southeast (F-SE) stores 5, the front southwest stores 3 and so on.

5.1.2 *Ordering of cells within a bunch.* In this paragraph, it will be shown how the cells of a bunch are ordered according to each basic shape. This ordering amounts to determine for each cell of the bunch which cell its *next* and *previous* pointers will point to. Table II describes the ordering chosen for each basic shape number.

Let (i, j) denote the cell corresponding to the i-th row and j-th column in Table II, $i = 0, \ldots, 11$ and $j = 1, \ldots, 8$. The ordering of the bunch's cells according to basic shape i is done by making the *next* pointer of cell (i, j) point to cell $(i, j+1)$ for $j = 1, \ldots, 7$ and *previous* pointer of cell (i, j) point to cell $(i, j-1)$ for $j = 2, \ldots, 8$. Pointers *previous* of cell $(i, 1)$ and *next* of cell $(i, 8)$ are treated separately since they are responsible for connecting the curve segment in the bunch to the main curve in the graph making a continuous curve. Pointer *previous* of cell $(i, 1)$ must point to the cell referenced by pointer *previous* of the cell which originated the bunch, while pointer *next* of cell $(i, 8)$ must point to the cell referenced by pointer *next* of the cell which originated the bunch.

Table II. Ordering of cells within a bunch according to the number of basic shape used.

	1st	**2nd**	**3rd**	**4th**	**5th**	**6th**	**7th**	**8th**
0	F-NE	B-NE	B-NW	F-NW	F-SW	B-SW	B-SE	F-SE
1	F-NE	F-SE	F-SW	F-NW	B-NW	B-SW	B-SE	B-NE
2	F-NE	B-NE	B-SE	F-SE	F-SW	B-SW	B-NW	F-NW
3	B-NW	F-NW	F-NE	B-NE	B-SE	F-SE	F-SW	B-SW
4	F-SW	B-SW	B-NW	F-NW	F-NE	B-NE	B-SE	F-SE
5	B-SE	B-NE	B-NW	B-SW	F-SW	F-NW	B-NE	B-SE
6	F-SW	F-NW	F-NE	F-SE	B-SE	B-NE	B-NW	B-SW
7	B-NW	B-SW	F-SW	F-NW	F-NE	F-SE	B-SE	B-NE
8	B-SE	F-SE	F-SW	B-SW	B-NW	F-NW	F-NE	B-NE
9	B-SE	F-SE	F-NE	B-NE	B-NW	F-NW	F-SW	B-SW
10	F-SW	B-SW	B-SE	F-SE	F-NE	B-NE	B-NW	F-NW
11	B-NW	B-NE	B-SE	B-SW	F-SW	F-SE	F-NE	F-NW

5.2 Hilbert curve in refinement

As seen in Section 3, the ordering of the cells of a new bunch occurs in the refinement procedure (Step 6). After the creation and linking of the cells of the bunch, the attribute *hilbertShapeNumber* and pointers *next* and *previous* are configured for each cell. This is done by Algorithm 17, where variable *cellNode* contains a reference for the cell which is being refined.

5.3 Hilbert curve in derefinement

When a bunch of cells is derefined, the cell which replaces the bunch must restore the attribute *hilbertShapeNumber* that the cell which originated the bunch had, so that if this cell is refined in some later iteration of the algorithm, the cells of the refined bunch will be ordered in the same way. This way, the mesh can be repeatedly

Algorithm 17 Ordering of the cells of a bunch by the modified Hilbert Curve

```
1: $i \leftarrow cellNode.hilbertShapeNumber$
2: for each bunch's cell node do
3:     set current cell's $hilbertShapeNumber$ according to row $i$ of Table I
4:     set current cell's $next$ pointer according to row $i$ of Table II
5:     set current cell's $previous$ pointer according to row $i$ of Table II
6: end for
```

refined and derefined, always having the cells ordered in the same way for a given mesh configuration. However, since the mesh cells do not store this information, it is necessary to create a mechanism to find the value of the Hilbert shape number of the originating cell from a combination of the Hilbert shape numbers of the cells which constitute the bunch. Function *getFatherHilbertShape()* in Algorithm 18 performs this task using Table I which gives the sum of Hilbert shape numbers for the cells in a bunch according to the basic shape used when ordering them.

One notices in Table I that there are no identical rows. Thus, the sequence of the Hilbert shape numbers of the bunch cells could be used as a way to identify the basic shape utilized in ordering the bunch's cells. On the other hand, one also notices that the sum of the Hilbert shape numbers of the bunch's cells are also different in every row, with the exception of rows 4, 10 and 11, whose sum equals 45 (see Table III). Therefore, instead of using a sequence of 8 numbers, the single number provided by this sum is used to identify most of the basic shapes; in the three cases where the sum is 45, one needs only to verify the Hilbert shape number of the second cell of the bunch (with respect to the ordering given by Table II), since it is different in any of the basic shapes 4, 10 and 11.

Table III. Sum of Hilbert shape numbers for the cells in a bunch according to each basic shape.

Basic shape	0	1	2	3	4	5	6	7	8	9	10	11
Sum	24	38	50	49	45	56	41	43	58	37	45	45

Algorithm 18 Returns the Hilbert's basic shape of a given bunch upon receiving the bunch's first cell as a parameter

```
procedure getFatherHilbertShape(firstBunchCell)
    $sum \leftarrow 0$
    $hilbertShape \leftarrow -1$
    $auxiliarCell \leftarrow firstBunchCell$

    for i $\leftarrow$ 1 to 8 do
        $sum \leftarrow sum + auxiliarCell \cdot hilbertShapeNumber$
        $auxiliarCell \leftarrow auxiliarCell \cdot next$
    end for

    if $sum == 24$ then
        $hilbertShape \leftarrow 0$
    else if $sum == 38$ then
        $hilbertShape \leftarrow 1$
    else if $sum == 50$ then
        $hilbertShape \leftarrow 2$
    else if $sum == 49$ then
        $hilbertShape \leftarrow 3$
    else if $sum == 45$ then
        if $firstBunchCell \cdot next \cdot hilbertShapeNumber == 10$ then
            $hilbertShape \leftarrow 4$
        else if $firstBunchCell \cdot next \cdot hilbertShapeNumber == 4$ then
            $hilbertShape \leftarrow 10$
        else
            $hilbertShape \leftarrow 11$
        end if
    else if $sum == 56$ then
        $hilbertShape \leftarrow 5$
    else if $sum == 41$ then
        $hilbertShape \leftarrow 6$
    else if $sum == 43$ then
        $hilbertShape \leftarrow 7$
    else if $sum == 58$ then
        $hilbertShape \leftarrow 8$
    else if $sum == 37$ then
        $hilbertShape \leftarrow 9$
    else
        $print('Error')$
    end if
    return $hilbertShape$
end procedure
```

6. APPLICATION ON CARDIAC ELECTROPHYSIOLOGY SIMULATION

Simulations of cardiac electrophysiology have become a valuable tool for the study and comprehension of the heart's bioelectric activity under normal and pathological conditions. These simulations are usually based on the bidomain model [J. Sundnes et al. 2006], which is a system of two partial differential equations (PDEs) coupled to a set of nonlinear ordinary differential equations (ODEs) describing the behavior of the membrane of cardiac cells. In spite of being currently the most complete description of the electrical activity of the heart, the numerical solution of the bidomain equations is a computational challenging task [R. W. Santos et al. 2004]. Usually the bidomain equations are reduced to the simpler monodomain model by considering that: the extracellular potential is constant; or tissue conductivity is isotropic; or the intracellular and extracellular conductivities have equal anisotropy ratios [J. Sundnes et al. 2006].

6.1 Monodomain Model

In cardiac tissue, the excitation wave spreads through the tissue because the cardiac cells are electrically coupled via special proteins called gap junctions. This phenomenon is mathematically described by a reaction–diffusion equation referred to as the monodomain equation, given by

$$\beta C_m \frac{\partial V_m}{\partial t} + \beta I_{ion}(V_m, \boldsymbol{\eta}) = \nabla \cdot (\boldsymbol{\sigma_m} \nabla V_m) + I_{stim} \tag{2}$$

$$\frac{\partial \boldsymbol{\eta}}{\partial t} = \boldsymbol{f}(V_m, \boldsymbol{\eta}) \tag{3}$$

where β is the surface-to-volume ratio of the cardiac cells, C_m is the membrane capacitance, V_m is the transmembrane voltage, I_{ion} is the density of the total ionic current which is a function of V_m and a vector of state variables $\boldsymbol{\eta}$, I_{stim} is a stimulus current and $\boldsymbol{\sigma_m}$ is the monodomain conductivity tensor. The tissue is assumed to be isolated along its boundaries, i.e. no–flux boundary conditions are imposed on V_m along all myocardial surfaces.

In this work the classical Luo–Rudy I (LRI) model [C. Luo and Yoram Rudy 1991] that describes the electrical activity in a general mammalian ventricular cell was considered to simulate the kinetics of I_{ion} in Eq. (3). In this mammalian ventricular model, I_{ion} is defined as the following sum of currents:

$$I_{ion} = I_{Na} + I_{si} + I_K + I_{K1} + I_{Kp} + I_b \tag{4}$$

where I_{Na} is the fast sodium current, I_{si} is the slow inward current, I_K is the time-dependent potassium current, I_{K1} is the time-independent potassium current, I_{Kp} is the plateau potassium current and I_b is time-independent background current. The LRI model is based on a set of 8 ODEs describing ionic currents and intracellular calcium concentration. For a full specification of the model and its ionic currents see [C. Luo and Yoram Rudy 1991].

6.2 Finite Volume model applied to monodomain

In this section we will make a brief description of the Finite Volume Method (FVM) applied to the monodomain equations. Details about the FVM applied to monodomain can be found in [Harrild and Henriquez 1997] and [Coudiere et al. 2009].

The reaction and diffusion part of the monodomain equations can be split by employing the Godunov operator splitting [J. Sundnes et al. 2006]. Each time step involves the solution of two different problems: a nonlinear system of ODEs

$$\frac{\partial V_m}{\partial t} = \frac{1}{C_m}\left[-I_{ion}(V_m, \eta_i) + I_{stim}\right] \tag{5}$$

$$\frac{\partial \eta_i}{\partial t} = f(V_m, \eta_i) \tag{6}$$

and a parabolic linear PDE

$$\frac{\partial V_m}{\partial t} = \frac{1}{\beta C_m}\left[\nabla \cdot (\boldsymbol{\sigma} \nabla V_m)\right] \tag{7}$$

Depending on the numerical method, the spatial discretization of the parabolic PDE results in a linear system of equations that have to be solved at each time step.

6.2.1 *Time discretization.* The time derivative present in Equation (7), which operates on V is approximated by an implicit first–order Euler scheme:

$$\frac{\partial V}{\partial t} = \frac{V^{n+1} - V^n}{\Delta t}, \tag{8}$$

where V^n represents the transmembrane potential at time t_n and Δt the increment at every step.

6.2.2 *Space discretization.* The diffusion term of Equation (7) needs to be spatially discretized. To do this we will consider the following relations:

$$J = -\sigma \nabla V \tag{9}$$

where J $(\mu A/cm^2)$ represents the density of the intracellular current flow and

$$\nabla \cdot J = -I_v. \tag{10}$$

In this expression, $I_v(\mu A/cm^3)$ is a volumetric current and corresponds the left side of Equation (7).

For simplicity, we will consider a tri-dimensional uniform mesh, consisting of cubes (called "Volume"). Situated in the center of each volume is a node and V is associated with each node of the mesh.

After defining the geometry of the mesh and the partitioning of the domain in control volumes, the FMV-specific equations can be presented. The Equation (10) can be integrated spatially over a cube specific, leading to:

$$\int_\Omega \nabla \cdot J da = -\int_\Omega I_v \, da. \tag{11}$$

applying the divergence theorem, we find that

$$\int_\Omega \nabla \cdot J da = \int_{\partial\Omega} J \cdot \vec{\eta}, \tag{12}$$

where $\vec{\eta}$ is the vector normal to the edge.

Finally, assuming that I_v represents an average value in each particular cube, and substituting in Eq (7), we have the following relationship:

$$\beta \left(C_m \frac{\partial V}{\partial t} \right) \Bigg|_{(i,j,k)} = \frac{-\int_{\partial\Omega} J \cdot \vec{\eta}}{h^3}, \tag{13}$$

where h^3 is the volume of the control cell and $\vec{\eta}$ represents the vector normal to the surface.

For the three-dimensional problem, formed by a uniform grid of cubes with face area h^2, the calculation of J can be split as the sum of the flows on the six faces:

$$\int_{\partial\Omega} J \cdot \vec{\eta} = h^2 \cdot \sum_{l=1}^{6} J f_l \tag{14}$$

where,

$$\begin{aligned} \sum_{l=1}^{6} J f_l &= J_{x_{i+1/2,j,k}} - J_{x_{i-1/2,j,k}} \\ &+ J_{y_{i,j+1/2,k}} - J_{y_{i,j-1/2,k}} \\ &+ J_{z_{i,j,k+1/2}} - J_{z_{i,j,k-1/2}}, \end{aligned} \tag{15}$$

The tensor $\sigma = \begin{pmatrix} \sigma_x & 0 & 0 \\ 0 & \sigma_y & 0 \\ 0 & 0 & \sigma_z \end{pmatrix}$ must be determined at the interfaces of the volume. For this, we use the harmonic mean:

$$\sigma_{x_{i+1/2,j}} = \frac{2\sigma_{x_{i,j}} \sigma_{x_{i+1,j}}}{\sigma_{x_{i+1,j}} + \sigma_{x_{i,j}}} \tag{16}$$

A similar reasoning can be used to calculate $\sigma_{x_{i-1/2,j,k}}$, $\sigma_{y_{i,j+1/2,k}}$, $\sigma_{y_{i,j-1/2}}$, $\sigma_{z_{i,j,k+1/2}}$ and $\sigma_{z_{i,j,k-1/2}}$.

The flows $J_{x_{m,n,o}}$, $J_{y_{m,n,o}}$ and $J_{y_{m,n,o}}$ are calculated in the faces $((m,n,o) = (i{+}1/2, j, k), (i{-}1/2, j, k), (i, j{+}1/2, k), (i, j{-}1/2, k), (i, j, k{+}1/2)$ or $(i, j, k{-}1/2))$ as follows:

$$J_{x_{m,n,o}} = \sigma_x(m,n,o) \frac{\partial V}{\partial x} \Bigg|_{(m,n,o)} \tag{17}$$

$$J_{y_{m,n,o}} = \sigma_y(m,n,o) \frac{\partial V}{\partial y} \Bigg|_{(m,n,o)} \tag{18}$$

$$J_{z_{m,n,o}} = \sigma_z(m,n,o) \frac{\partial V}{\partial z} \Bigg|_{(m,n,o)} \tag{19}$$

6.2.3 *Adaptive non-uniform mesh (ALG).* In this section we present the application of the FVM using an adaptive non- uniform mesh, in this case ALG. For this, we will approximate the partial derivatives of V on the interfaces using the following finite difference scheme, considering uniform discretizations in space

$(\Delta x = \Delta y = \Delta z = h)$:

$$\left.\frac{\partial V}{\partial x}\right|_{(i+1/2,j,k)} = \frac{V_{i+1,j,k} - V_{i,j,k}}{h} \tag{20}$$

The equations for y and z can be obtained similarly.

Rearranging and replacing the discretizations of the equations (8) and (14) in Equation (13) and decomposing the operators as described by the equations (5), (6) and (7) we have:

$$C_m \frac{V^*_{i,j,k} - V^n_{i,j,k}}{\Delta t} = \frac{h^2 \cdot \sum_{l=1}^{6} J{f^*}_l}{\beta h^3} \tag{21}$$

$$C_m \frac{V^{n+1}_{i,j,k} - V^*_{i,j,k}}{\Delta t} = -I_{ion}(V^*_{i,j,k}, \boldsymbol{\eta}^n) \tag{22}$$

$$\frac{\partial \boldsymbol{\eta}^{n+1}}{\partial t} = f(\boldsymbol{\eta}^n, V^*, t) \tag{23}$$

where:

$$J^*_{x_{i+1/2,j}} = \sigma_{x_{i+1/2,j,k}} \frac{V^*_{i+1,j,k} - V^*_{i,j,k}}{h}, \tag{24}$$

Eq. (24) represents the flow of the cell (i, j, k) by the right face of the cube to the neighbor cell, n is the current step, $*$ is an intermediate step and $n+1$ is the next time step. Equations for $J^*_{x_{i-1/2,j,k}}$, $J^*_{y_{i,j+1/2,k}}$, $J^*_{y_{i,j-1/2,k}}$, $J^*_{z_{i,j,k+1/2}}$ and $J^*_{z_{i,j,k-1/2}}$ can be obtained similarly. Developing all equations, we can now define the time advance formula for the interior points of each volume:

$$\begin{aligned}(\sigma_{x_{i+1/2,j,k}} + \sigma_{x_{i-1/2,j,k}} + \sigma_{y_{i,j+1/2,k}} + \sigma_{y_{i,j-1/2,k}}\sigma_{x_{i,j,k+1/2}} + \sigma_{x_{i,j,k-1/2}} + \alpha)V^*_{i,j,k} +& \\ \sigma_{y_{i,j-1/2,k}} V^*_{i,j-1,k} -& \\ \sigma_{x_{i+1/2,j,k}} V^*_{i+1,j,k} -& \\ \sigma_{z_{i,j,k+1/2}} V^*_{i,j,k+1} -& \\ \sigma_{y_{i,j+1/2,k}} V^*_{i,j+1,k} -& \\ \sigma_{x_{i-1/2,j,k}} V^*_{i-1,j,k} = V^t_{i,j,k} * \alpha,&\end{aligned}$$

where $\alpha = (\beta C_m h^2)/\Delta t$.

Algorithm 19 describes the steps used for the numerical resolution of monodomain model. As can be seen, we have to reassemble the monodomain matrix at each time step if a refinement or derefinement operation has been performed in that step. In this example application, the criteria used for refinement and derefinement are based on the flux across the interface of neighboring cells. That is, if the absolute value of the flux is larger than a predefined refinement threshold ($reft$), the program chooses to refine this cell, whereas if the absolute value of the flux of all four cells of a bunch is less than an derefinement threshold ($dreft$), the program chooses to derefine the bunch. For the monodomain application, the values for the refinement and derefinement thresholds where empirically found.

Algorithm 19 Steps used for the numerical resolution of monodomain model

```
1: set cell model initial conditions;
2: assemble the monodomain matrix (Linear system form PDE);
3: while $t < final\_t$ do
4:     update cell Model state vector;
5:     solve cell model;
6:     solve linear system (PDE) via conjugate gradient method;
7:     refine-derefine
8:     reassemble the monodomain matrix if needed;
9:     $t = t + dt$
10: end while
```

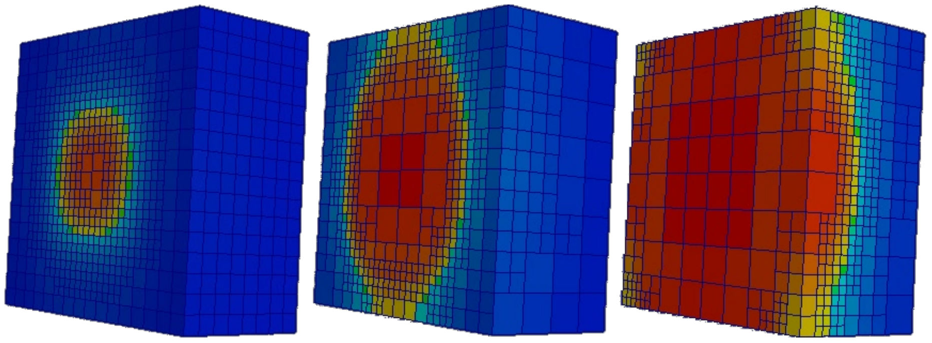


Fig. 17. Simulation using ALG adaptivity scheme. Result of the simulated electric wave propagation (distribution of transmembrane potential V) in a ventricular tissue.

Figure 17 shows the simulation results using ALG adaptive mesh. We can clearly see that the region delimited by the wavefront has a spatial discretization smaller than the rest of the tissue. The colors represents the distribution of transmembrane potential V.

REFERENCES

Burgarelli, D. Modelagem computacional e simulação numérica adaptativa de equações diferenciais parciais evolutivas aplicadas a um problema termoacústico. Ph.D. thesis, Tese de doutorado, PUC-Rio, Rio de Janeiro, Brasil, 1998 (in Portuguese).

Burgarelli, D., Kischinhevsky, M., and Biezuner, R. J. 2006. A new adaptive mesh refinement strategy for numerically solving evolutionary pde's. *J. Comput. Appl. Math. 196*, 115–131.

C. Luo and Yoram Rudy. 1991. A model of the ventricular cardiac action potential. depolarization, repolarization, and their interaction. *Circ Res 68,* 6, 1501–26.

Coudiere, y., Pierre, C., and Turpault, R. 2009. A 2d/3d finite volume method used to solve the bidomain equations of electrocardiology. In *proceedings of Algoritmy.* 1–10.

Gamezo, V., Khokhlov, A., and Oran, E. 2005. Three-dimensional delayed-detonation model of type ia supernovae. *The Astrophysical Journal 623*, 337.

Harrild, D. and Henriquez, C. 1997. A finite volume model of cardiac propagation. *Annals of biomedical engineering 25,* 2, 315–334.

J. Sundnes, G. T. Lines, X. Cai, B. F. Nielsen, K. A. Mardal, and A. Tveito. 2006. *Computing the Electrical Activity in the Heart.* Springer.

Ji, H., Lien, F., and Yee, E. 2008. A robust and efficient hybrid cut-cell/ghost-cell method with adaptive mesh refinement for moving boundaries on irregular domains. *Computer Methods in Applied Mechanics and Engineering 198,* 3-4, 432–448.

Penner, J., Andronova, N., Oehmke, R., Brown, J., Stout, Q., Jablonowski, C., Leer, B., Powell, K., and Herzog, M. 2007. Three dimensional adaptive mesh refinement on a spherical shell for atmospheric models with lagrangian coordinates. In *Journal of Physics: Conference Series*. Vol. 78. IOP Publishing, 012072.

R. W. Santos, G. Plank, S. Bauer, and E. J. Vigmond. 2004. Parallel multigrid preconditioner for the cardiac bidomain model. *IEEE Trans Biomed Eng 51,* 11, 1960–1968.

Tavakoli, R. 2008. Cartgen: Robust, efficient and easy to implement uniform/octree/embedded boundary cartesian grid generator. *International Journal for Numerical Methods in Fluids 57,* 12, 1753–1770.