Abstract

The data structures used to model meshes for solving problems by finite element methods is based on different arrays. In these arrays information is stored related to, among other components, nodes, edges, faces, tetrahedral and connectivity. These structures provide optimum results which, in many cases, incur additional programming. In adaptively solving problems, the meshes undergo refinement/derefinement processes, to improve the numeric solution with each step. These processes produce new elements and eliminate others, so the arrays should reflect the state of the mesh in each of these steps. Using traditional language, memory should be pre-assigned at the outset of the program, so it is only required to estimate the changes taking place in the mesh. In the same respect, it was necessary to compact the arrays to recover space from erased elements. With the advent of languages such as C, memory can be assigned dynamically, resolving most of the problem. However, arrays are costly to maintain, as they require adapting the mesh treatment to the data model, and not inversely. The object-oriented program suggests a new focus in implementing data structures to work with meshes. The classes create data types that may be adjusted to the needs of each case, allowing each element to be modeled on an independent, exclusive basis. Inheritance and encapsulation enable us to simplify the programming tasks and increase code reuse. We propose a data structure based on objects for treating meshes. Finally, we present an implementation of a local refinement algorithm based on the subdivision of tetrahedra in 8-sub-tetrahedra and some experiments.

Keywords: 3-D triangulations, unstructured grids, nested meshes, adaptive refinement, object oriented methods, data structures, finite element method.

1 Introduction

Most programs currently using the finite element method rely on adapted techniques based on a committed error estimation with our numerical solution, or at least on reliable error indicators that specify the elements that should be refined or derefined in the mesh.

In adaptive mesh generation we may consider two different aspects: domain discretization in accordance with its geometry or numerical solution. There are many ways to approach these aspects. We first need to consider whether the meshes are structured or unstructured. In this respect, the use of unstructured meshes clearly provides more flexibility when meshing complex geometries using an optimum number of nodes. In this case, the classic methods of obtaining three-dimensional triangulations is based mainly on advancing front algorithms [1] or in those based on Delaunay triangulation [2, 3]. Once the domain geometry has been discretized, the mesh should be adapted to the specific numerical solution. This process involves the introduction (refinement) or elimination (derefinement) of nodes in the current mesh. The changes may alter the current mesh locally or globally, depending on the method of triangulation chosen. Different refinement strategies have been developed for 2-D triangulations, and they have been generalized to 3-D. If we choose a refinement that affects the current mesh locally, another question is raised: nested or unnested meshes? In this case the answer is not clear. We may obtain families of nested mesh sequences in a minimal CPU time. Furthermore, the multigrid method can be more easily applied to solve the equations system associated with the problem. We may also automatically control the smoothness and degeneration of the mesh, as well as maintaining the defined surfaces in the domain, according to the characteristics of the initial mesh. If the domain has a complex geometry, a good way to proceed involves obtaining the initial mesh with an unstructured mesh generator and, subsequently, applying a nested local mesh refinement and derefinement technique using an error indicator appropriate to the problem. If we attempt to solve a developing problem, we may automatically approximate any initial solution defined in the domain. With the refinement and derefinement technique, we obtain optimum piecewise interpolation capable of approximating this solution with desired precision. In general, this technique can be applied to any function defined in the domain discretely or analytically.

With these ideas, techniques were developed previously which were adaptable to 2-D and obtained good results in different stationary and evolutive problems, see for example [4, 5, 6]. In these studies Rivara's 4-T local refinement algorithm was used; all the triangles must be refined, bearing in mind the error indicator, they are divided into four sub-triangles using a new node in the centers of its sides and bringing together the node introduced in the biggest side with the opposing vertex and the other two new nodes. Choosing the particular refinement algorithm is very important, as the derefinement algorithm may be understood as the inverse of the refinement algorithm. Rivara's 4-T refinement algorithm contains good properties in terms of mesh smoothness and degeneration. In addition, the number of possibilities that appear in the relation between a father element and sons is less than with other refinement al-

gorithms in 2-D, after ensuring the conformity of the mesh. Thus, it would be more complicated to develop a derefinement algorithm, coupled with the local refinement algorithm as proposed in [7]; all the triangles that must be refined, maintaining the error indicator, they are divided into four subtriangles by introducing a new node in the centers of the sides and joining them to each other.

In 3-D, we have a different problem. Paradoxically, the extension of an adaptive algorithm that may be simpler than another in 2-D, may not be simpler in 3-D. Thus, in the refinement algorithms developed in 3-D we note that they are based on the tetrahedral bisection [8, 9, 10] and those that use the 8-subtetrahedral subdivision [11, 12, 13]. The algorithm developed in [10] may be understood as the generalization to 3-D of Rivara's 4-T algorithm, which is itself based on the bisection of the triangle by its largest side. The problem in this extension to 3-D is the increased number of possible cases in which the tetrahedron may be divided, maintaining the possible differences of the 4-T divisions in their four faces, during the process of conformity of the mesh. However, the algorithms analyzed in [11, 12, 13], which themselves generalize to 3-D the partition into four sub-triangles as proposed in [7], are more simple due to the number of possible partitions in a tetrahedron is much less than the case of the generalization of the 4-T algorithm. Furthermore, as mesh quality is ensured in all these cases, we have chosen to implement a version of the algorithm which uses the subdivision in 8-subtetrahedra. This algorithm will be considered in section 3 of this paper and section 4 is devoted to the its implementation.

In section 5, applications of the refinement algorithm on three-dimensional meshes, generated by a version of Delaunay triangulation method presented in [3], are carried out. This triangulation method is widely accepted due to the quality of the meshes produced. However, it presents serious problems, particularly in 3-D, because of the rounding errors which occur in the computer and which are exacerbated by problems inherent in this method of triangulation itself. In [3] a procedure is presented for constructing a three dimensional triangulation similar to Delaunay technique which solves these problems. As Delaunay triangulation is constructed on a set of points located on the boundary and inside the domain, it may be that the resulting mesh does not contain all the main edges and faces defined in the boundary or interior domain. This aspect constitutes an open problem and has been studied by several authors, although the solutions offered are complex. The size of the mesh considered will depend on the complexity of the problem and quality of the solution sought. When the meshes are fine we are faced with two problems: on the one hand, the space required to store the mesh; on the other, the time needed to process the information.

These problems are generally dealt with in everyday use. However large the data stored, more swift the process, whilst for greater economy of information, the processing will be more complex. One possible solution that satisfies these requirements would entail an appropriate organization of the information, thus minimizing the storage space and also the information processing.

The data structures traditionally used in mesh problems are based on the different arrays that contain mesh information: nodes, edges, faces, tetrahedra, connectivity,

genealogy, etc. Generally, in languages such as FORTRAN it is important to oversize these arrays to anticipate mesh changes. When refinements are necessary, the increase in the number of elements must be estimated. When derefinements are carried out, the space of the eliminated elements should be recovered by compacting the arrays.

All this memory work may lead to significant time wastage in programming, time better spent on other tasks. Recovering space also takes time, depending on programming efficiency.

Some problems are solved with the development of languages such as C. With C memory may be used dynamically: when memory is needed, it is obtained from the system, and it may be returned when it is not needed and thus used again. The recovery and compacting of memory is left to the operating system, so the programming may be concentrated on other aspects of the problem.

Structures are another advantage of C. They allow for a clearer organization of information, thus facilitating the programming. In the structures, information for each element is grouped, independently of the type of data being treated. This does not save used memory space for storage, but does provide more clarity in the programs.

The pointers are another tool. In C objects can be referenced indirectly using direction finders to the positions the memory takes up. We will see that this provides considerable savings when passing information between modules and increases the efficiency of information organization.

Considerable progress has been made with C++ that extends the concept of structure to that of class. A class contains all the operations which can be carried out with it and also element information. In other words, a type of data based on the element is established.

Furthermore, the program aimed at objects introduces the concept of inheritance. A class may be seen as an heir to another, so that it will possess all the properties of its predecessor, plus the new ones that are its own. This permits us to develop hierarchies of classes, and continue creating increasingly complex modules from more simple ones.

Another interesting concept is encapsulation. In each class we may define what parts are accessible to the outside and which are restricted.

With these characteristics a class may be considered a black box which provides interfaces with the remaining modules, whilst its inner workings are absolutely private. The number of programming errors is reduced, as only class-authorized operations may be carried out. Meanwhile code reuse increases.

2 Hierarchy of Classes

Based on the classes and C++, a hierarchy has been defined for modelling the different elements that make up a mesh, and gather together the characteristics of a refinement/derefinement process when solving adaptive problems. The general structure

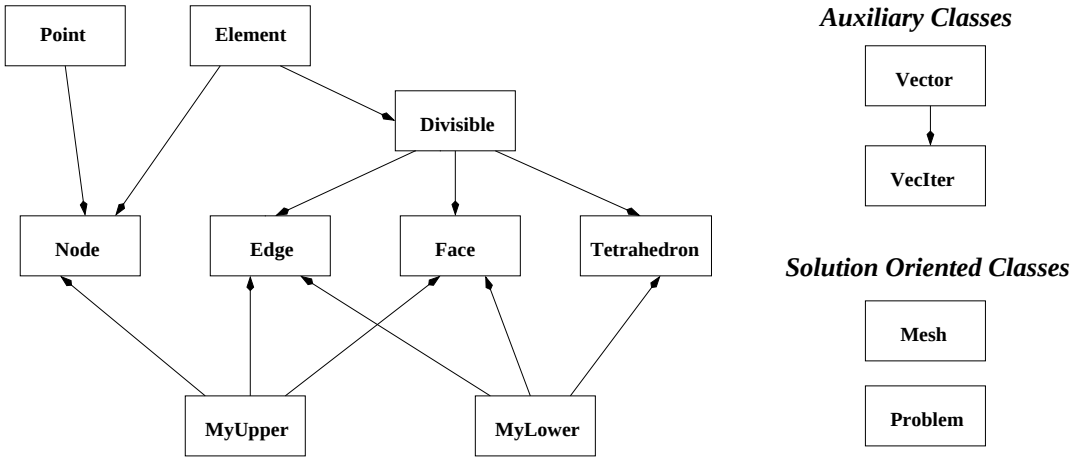


Figure 1: Hierarchy of classes

may be seen in figure 1.

We begin by providing a detailed description of the auxiliary classes, used by the rest of the modules for their internal tasks. Then we will consider the classes used for modeling the mesh elements. Finally we will present the classes oriented at solving problems.

a) *Auxiliary classes*. A so-called *Vector* class has been defined, which is an array with certain peculiarities. The most striking is that it does not contain duplicate elements and its elements are always pointers or references to objects. There are operations programmed for addition and extraction of elements, addition of contents of one array over another, and accesses to the array elements by index.

Dynamic memory is also carried out in this class, by borrowing and returning to the system. The rest of the classes use this class to maintain references to other objects.

From Vector class we can also define the *VecIter* class. It inherits all the Vector characteristics, and introduces operations that allow us to carry out revisions of the array elements, as well as simpler recovery methods.

b) *Point*. In this class the basic properties are defined of a point in space and some operations that may be carried out with it. It contains the coordinates (x,y,z) of the point, and the operations for addition and subtraction of coordinates, multiplication by a constant and the distance between two points.

c) *Element*. This will be the basic class for all the elements of the mesh. It is very simple, only containing a single property called *Reference*, used in all the objects that make up the mesh.

d) *Divisible*. This class is inherited from *Element*, and is the antecedent to all those that model objects susceptible to be divided. It contains genealogical references of the elements, just as a parent is the one and the sons are the elements into which the being is divided. In the previously marked processed to mesh element division, this class ensures elements are not marked twice erroneously, and provides information on the

current state of each one.

e) *MyUpper*. For each given element, this class maintains the object references that compose it, for example, for a node it indicates the edges contained.

f) *MyLower*. Contrary to the previous example, the object references that make up a given are stored.

g) *Node*. This class is inherited from *Point*, *Element* and *MyUpper*. It contains the data necessary for modeling a mesh node. At any time, and through the data contained in the parent classes it is possible to access the rest of the elements contained in a certain node.

h) *Edge*. This class is inherited from *MyLower*, *Divisible* and *MyUpper*. As with the nodes, references are stored as to which elements (faces) belong, in addition to references to the nodes that form it. The reference of the possible node that divides the edge is also stored in a refinement process. Another implemented procedure returns the length of the edge.

i) *Face*. This has the same inheritance as the edge. References are stored to possible interior edges which may result from a process of division of the face.

j) *Tetrahedron*. This class is inherited from *MyLower* and *Divisible*. It contains references to the faces that form it. Furthermore some indicators are stored for tetrahedra and dependents of the problem under consideration, as well as whether they are refined or not and the initial value of a solution in that element.

k) *Classes oriented to solutions*. Two classes have been defined, directly related to solving problems through refinement and derefinement. One is the *Mesh* class, which contains a list of references to node, edges, faces and tetrahedra, all related to each other, that form the mesh. In this module a refinement algorithm has been used based on the subdivision of the tetrahedra in eight sub-tetrahedra. To carry out this refinement different processes have been programmed to carry out the conformity of the mesh. Subsequently the algorithm will be described in detail. The other class is the *Problem* class that contains the procedures of information exchange with other modules for problem solving. These procedures include reading and writing in index files with distinct formats and generating mesh information in different data structures used by other programs. In the *Problem* class there are lists of references to node, edges, faces and tetrahedra. These lists are not merely for reference purposes, but objects in their own right. From these lists an object is formed of the mesh class copying references, so that the objects are only found once in memory, but may be referenced from many elements. The data transferences which are carried out between modules are references, that is, pointers, thus considerable time and memory consumption are saved. Refinement and derefinement processes are controlled in this class, as well as the transfer of information from problem resolution to the mesh in order to carry out a new refinement.

3 Refinement Algorithm

We propose a refinement algorithm based on the 8-subtetrahedron subdivision developed in [13]. Consider an initial triangulation τ_1 of the domain given by a set of n_1 tetrahedra $t_1^1, t_2^1, ..., t_{n_1}^1$. Our goal is to build a sequence of m levels of nested meshes $T = \{\tau_1 < \tau_2 < ... < \tau_m\}$, such that the level τ_{j+1} is obtained from a local refinement of the previous level τ_j. The error indicator ϵ_i^j will be associated to the element $t_i^j \in \tau_j$. Once the error indicator ϵ_i^j is computed, such element must be refined if $\epsilon_i^j \geq \theta \epsilon_{\max}^j$, being $\theta \in [0, 1]$ the refinement parameter and $\epsilon_{\max}^j$, the maximal value of the error indicators of the elements of τ_j. From a constructive point of view, initially we shall obtain τ_2 from the initial mesh τ_1, attending to the following considerations:

a) *8-subtetrahedron subdivision*. A tetrahedron $t_i^1 \in \tau_1$ is called of *type I* if $\epsilon_i^1 \geq \gamma \epsilon_{\max}^1$. Later, this set of tetrahedra will be subdivided into 8 subtetrahedra as Figure 2(a) shows; 6 new nodes are introduced in the middle point of its edges and each one of its faces are subdivided into four subtriangles following the division proposed by Bank [7]. Thus, four subtetrahedra are determinated from the four vertices of t_i^1 and the new edges. The other four subtetrahedra are obtained by joining the two nearest opposite vertices of the octohedron which result inside t_i^1. This simple strategy is that proposed in [13] or in [11], in opposite to others based on afin transformations to a reference tetrahedron, as that analysed in [12] which ensures the quality of the resulting tetrahedra. However, similar results were obtained by Bornemann et al. [11] with both strategies in their numerical experiments. On the other hand, for Lohner and Baum [13], this choice produces the lowest number of distorted tetrahedra in the refined mesh. Evidently, the best of the three existing options for the subdivision of the inner octohedron may be determined by analysing the quality of its four subtetrahedra, but this would augment the computational cost of the algorithm.

Once the *type I* tetrahedral subdivision is defined, we can find neighbouring tetrahedra which may have 6, 5, ..., 1 or 0 new nodes introduced in their edges that must be taken into account in order to ensure the mesh conformity. In the following we analyse each one of these cases. We must remark that in this process we only mark the edges of the tetrahedra of τ_1 in which a new node has been introduced. The corresponding tetrahedron is classified depending on the number of marked edges. In other words, until the conformity of τ_2 is not ensured by marking edges, this new mesh will not be defined.

b) *Tetrahedra with 6 new nodes*. Those tetrahedra that have marked their 6 edges for conformity reason, are included in the set of *type I* tetrahedra.

c) *Tetrahedra with 5 new nodes*. Those tetrahedra with 5 marked edges are also included in the set of *type I* tetrahedra. Previously, the edge without new node must be marked.

d) *Tetrahedra with 4 new nodes*. In this case, we mark the two free edges and it is considered as *type I*.

Proceeding as in (b), (c) and (d), we improve the mesh quality and simplify the algorithm considerably due to the global refinement defined in (a) by the error indicator.

One may think that this procedure can augment the refined region, but we must take into account that only 1 or 2 new nodes are introduced from a total of 6. Note that this proportion is less or equal to that arising in the 2-D refinement with the 4-T Rivara algorithm, in which the probability of finding a new node introduced in the longest edge of a triangle is $1/3$. This fact is accentuated in the proposed algorithm as its generalization in 3-D.

e) *Tetrahedra with 3 new nodes*. In this case, we must distinguish two situations:

e.1) If the 3 marked edges are not located on the same face, then we mark the others and the tetrahedron is introduced in the set of *type I* tetrahedra. Here, we can make the previous consideration too, if we compare this step with other algorithms based on the bisection by the longer edge.

In the following cases, we shall not mark any edge, i.e., any new node will not be introduced in a tetrahedron for conformity. We shall subdivide them creating subtetrahedra which will be called *transient subtetrahedra*.

e.2) If the 3 marked edges are located on the same face of the tetrahedron, then 4 transient subtetrahedra are created as Figure 2(b) shows. New edges are created by connecting the 3 new nodes one another and these with the vertex opposite to the face containing them. The tetrahedra of τ_1 with these characteristics will be inserted in the set of *type II* tetrahedra.

f) *Tetrahedra with 2 new nodes*. Also here, we shall distinguish two situations:

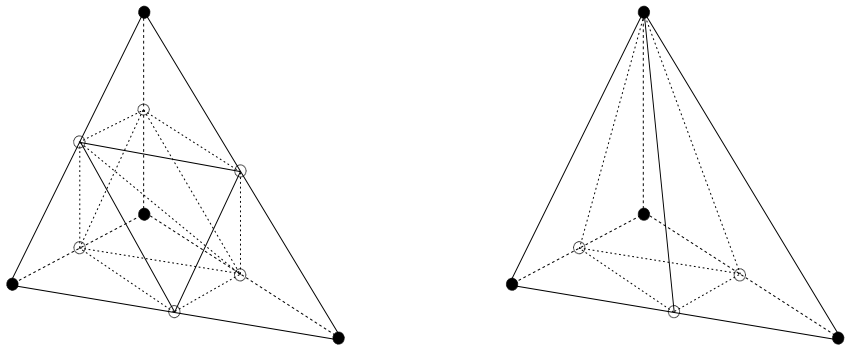
f.1) If the two marked edges are not located on the same face, then 4 transient subtetrahedra will be constructed from the edges conecting both new nodes and these with the vertices opposite to the two faces which contain each one of them. This tetrahedra are called *type III.a*; see Figure 2(c).

f.2) If the two marked edges are located on the same face, then 3 transient subtetrahedra are generated as Figure 2(d) shows. The face determinated by both marked edges is divided into 3 subtriangles, connecting the new node located in the longest edge with the vertex opposite and with the another new node, such that these three subtriangles and the vertex opposite to the face which contains them define three new subtetrahedra. We remark that from the two possible choices, the longest marked edge is fixed as reference in order to take advantage in some cases of the properties of the bisection by the longest edge. These tetrahedra are called *type III.b*.

g) *Tetrahedra with 1 new node*. Two transient subtetrahedra will be created as we can see in Figure 1(e). The new node is connected to the other two which are not located in the marked edge. This set of tetrahedra is called *type IV*.

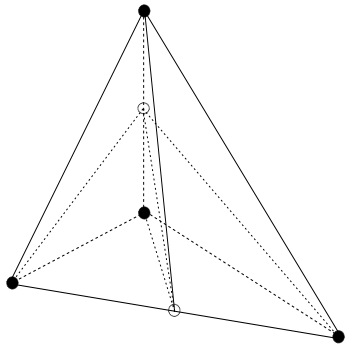
h) *Tetrahedra without new node*. These tetrahedra of τ_1 are not divided and they will be inherit by the refined mesh τ_2. We call them *type V* tetrahedra; see Figure 2(f).

This classification process of the tetrahedra of τ_1 is carried out by marking their edges simply. The mesh conformity is ensured in a local level analysing the neighbourhood between the tetrahedra which contain a marked edge by an expansion process that starts in the *type I* tetrahedra of paragraph (a). Thus, when the run along this set of *type I* tetrahedra is over, the resulting mesh is conformal and locally refined.

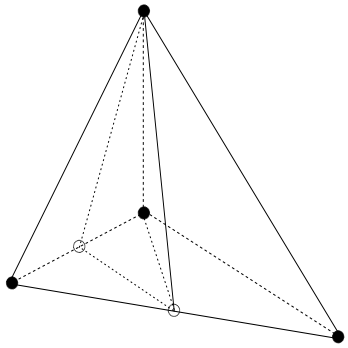


(a) *Type I*

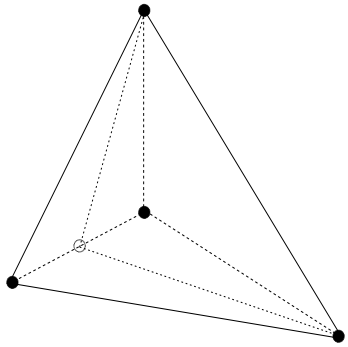
(b) *Type II*



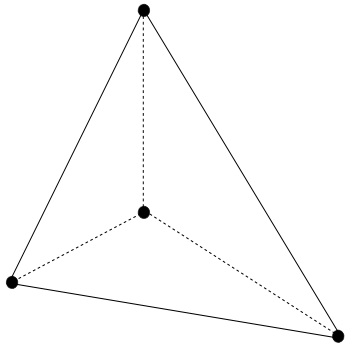
(c) *Type III.a*



(d) *Type III.b*



(e) *Type IV*



(f) *Type V*

Figure 2: Subdivision classification of a tetrahedron in function of the new nodes (white circles).

Moreover, this is a low computational cost process, since the local expansion stops when we find tetrahedra whose edges have not to be marked. Implementations details will be discussed on next section.

Generally, when we want to refine the level τ_j in which there already exist transient tetrahedra, we shall perform it in the same way as from τ_1 to τ_2, except for the following variation: if an edge of any transient tetrahedron must be marked, due to the error indicator or even to conformity reasons, then all the transient tetrahedra are eliminated from their parent (deleting process), all the parent edges are marked and this tetrahedron is introduced into the set of *type I* tetrahedra. We must remark that it will be only necessary to define a variable which determines if a tetrahedron is transient or not.

4 Algorithm Implementation

Algorithm development will contain, basically, two sequential revisions of the mesh. In the first, transient tetrahedra are studied and in the second marked non-transient tetrahedra.

In the first step two types of tetrahedra will be selected:

- the transients marked for refinement
- the transients with a neighboring tetrahedron by any of its edges which are refinable and non-transient (if it were transient the previous point would be selected)

In both cases, the selected parent tetrahedron will be called Type I to proceed to its division. The first point is based on the definition of the algorithm. In the second case there is an anticipation of what the algorithm is going to produce. As there is a marked, non-transient neighbor, it will be Type I, so all its edges should be marked, and at least one of these marks should be on the tetrahedron under consideration, which is transient, so it should be divided by conformity, which is not allowed, but rather it will be the father of the divided. This is what is selected in the second case.

Once the tetrahedra are marked, an expansion to conform the mesh is generated. A recursive process takes place, in which each step is studied, first, as to whether the tetrahedron is Type I, or whether it should be converted to Type I. If this is the case, for each edge which is still unmarked, a list of neighboring tetrahedra is created by the edge, marked edge, and for each tetrahedron a similar process is carried out. The following is a pseudo-code:

```
Main Process
  for every tetrahedron marked to be refined do
   Study (tetrahedron)

Study(Tetrahedron t)
 Drop_Inner_Division(t)
```

```
  if t have 6 marks return;
  if t is marked to be refined then
    Mark_All_Edges(t)
  else if t have 4 or 5 marks
    Mark_All_Edges(t)
  else if t have 3 marks not in the same face then
    Mark_All_Edges(t)

Mark_All_Edges(Tetrahedron t)
  for every edge of t do
    if edge is not marked then
      Mark_Edge(edge)

Mark_Edge(Edge a)
  for every tetrahedron of a do
    Drop_Inner_Division(tetrahedron)
  Make one mark in a
  for every tetrahedra of a do
    Study(tetrahedron)

Drop_Inner_Division(Tetrahedron t)
  if t is divided into 8 tetrahedra or not divided then
    return
  Remove inner tetrahedra of t
  for every face of t do
    Drop_Face_Division(face)

Drop_Face_Division(Face f)
  if f is divided into 4 faces or not divided then
    return
  For every tetrahedron of f do
    Drop_Inner_Division(tetrahedron)
  Remove division of f
```

As we can see, there are two stops criteria: the first is the *Study* process, when no marks should be made in a tetrahedron because it is adjusted to one of the types specified in the algorithm. The second is carried out after studying all the edges of a tetrahedron in the *Mark_All_Edges* process. The expansion process involves eliminating transients elements. Each time we study an element, and this is divided, its subdivision is eliminated, and the all the divisions of those neighboring the faces, since by carrying out a new marking will lead to different internal partitions to those already existing. The elimination process takes place with another revision of the tetrahedron under study toward all its neighbors, stopping when we have non-divided tetrahedra, or divided permanently in 8 sub-tetrahedra.

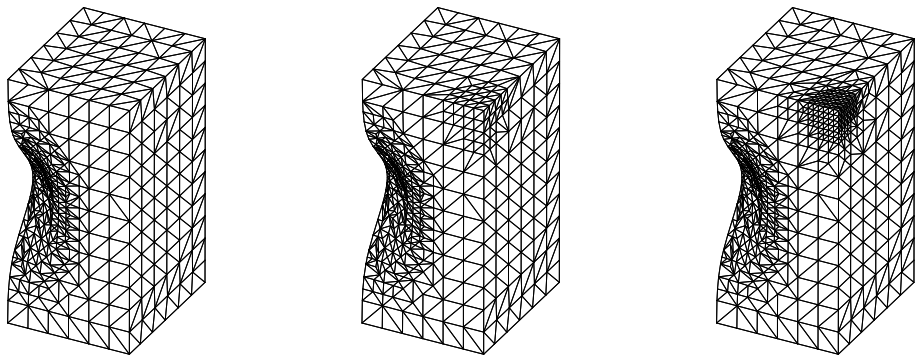
Once the expansion process is completed, we have a conforming mesh, and can begin to partition marked elements. The mesh tetrahedra are revised, the division of their edges and faces is carried out and new elements joined.

The second mesh revision will only study the tetrahedra that should be refined due to the numerical solution of the problem. All these tetrahedra will be non-transient, as marked transient ones have already been eliminated. This revision is similar to the process of the transients ones: tetrahedra are studied and marked, then divided and joined. The difference is that there it is no necessary to eliminate internal tetrahedral division (as we are not working with the parent of any element).

The algorithm and partition processes are programmed in the *Mesh* class. The objects created in the original mesh belong to the *Problem* class, while in each step of the refinement process references to the original objects are used. When an object is divided by *Mesh* class, its reference is eliminated, but not the object itself which does not belong to this class. The objects created are internal to others, they belong to the parent, and pass their references on to the *Mesh* class. As we can see, this class works by inserting and erasing references to objects in its lists, but it never creates or destroys any object, since that would be utilized in a subsequent step. When a satisfactory solution is obtained, the mesh class will be responsible for eliminating all the objects and returning memory to the system.

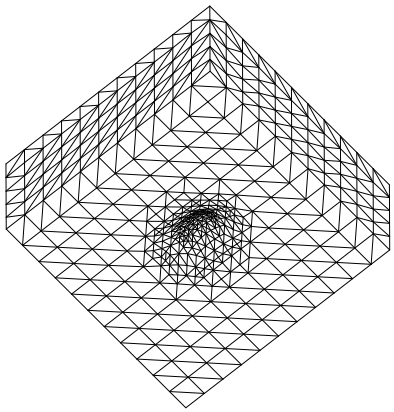
5 Numerical Experiments

The first experiement is related to a mesh which consists of 5072 tetrahedra and 1140 nodes. Here the refinement criteria is based on the distance from the gravity center of the tetrahedron to a corner of the domain.

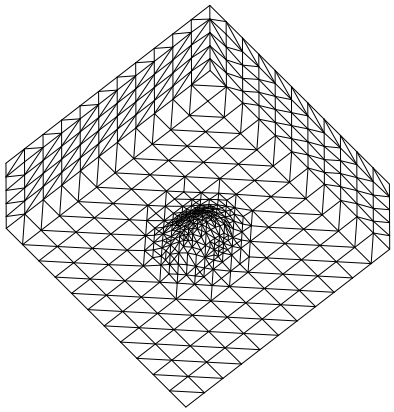


(a) τ_1 (b) τ_2 (c) τ_3

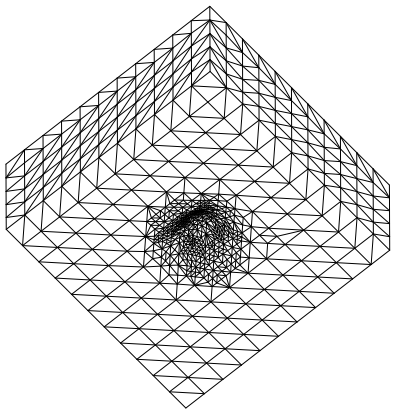
Figure 3: First experiment of the refinement algorithm; (a) initial mesh, (b) and (c) resulting meshes after 1 and 2 refinement steps, respectively.



(a) τ_1



(b) τ_2



(c) τ_3

Figure 4: Second experiment; (a) initial mesh, (b) and (c) refined meshes.

In figure 3 we present the resulting meshes after two steps of the refinement algorithm. The first one contains 5386 tetrahedral and 1201 nodes, while the second refinement yields a mesh with 6270 tetrahedral and 1433 nodes.

Figure 4 represents another mesh refined using our algorithm. We have began with a mesh of 5272 tetrahedra and 1229 nodes, obtained by the mesh generator developed by the authors in [14]; see figure 4(a). It has been refined according to an error indicator related to a wind field modelling using the finite element method. Two refinements has been computed. The first refined mesh, which contains 5408 tetrahedra and 1256 nodes, is shown in figure 4(b). The last one corresponds to figure 4(c) with 6696 tetrahedra and 1520 nodes. In this figure, only the lower surface and two vertical walls have been drawn in order to observe the local refinement around the mountain.

6 Conclusions

In this paper, some aspects of a 3-D mesh refinement algorithm have been presented. The class hierarchy is a robust tool for implementing the structure of meshes. New properties for elements were directly added when they were needed. Due to programming requirements, we consider from simpler characteristics to more complex ones. The implementation of the algorithm using the class hierarchy has reached the proposal aims: low computational cost and minimal memory requirements.

On the other hand, the refinement algorithm has interesting properties about quality and degeneration of meshes after many refinement steps. It has been properly applied in 3D-meshes generated by the version of Delaunay trangulation proposed in [3].

Finally, in future works we will develope the derefinement algorithm associated to the refinement one presented in this paper.

Acknowledgements

This work has been partially supported by the MCYT of Spanish Government and FEDER, grant contract REN2001-0925-C03-02/CLI. The authors acknowledge Dr. David Shea for editorial assistence.

References

[1] R. Lohner, P. Parikh, *"Three-dimensional grid generation by advancing front method"*, Int. J. Num. Meth. Fluids, 8, 1135-1149, 1988.

[2] P.L. George, F. Hecht, E. Saltel, *"Automatic mesh generation with specified boundary"*, Comp. Meth in Appl. Mech and Eng., 92, 269-288, 1991.

[3] J.M. Escobar, R. Montenegro, *"Several aspects of three-dimensional delaunay triangulation"*, Advances in Engineering Software, 27(1/2), 27-39, 1996.

[4] L. Ferragut, R. Montenegro, A. Plaza, *"Efficient refinement/derefinement algorithm of nested meshes to solve evolution problems"*, Comm. Num. Meth. Eng., 10, 403-412, 1994.

[5] R. Montenegro, A. Plaza, L. Ferragut, I. Asensio, *Application of a nonlinear evolution model to fire propagation*, Nonlinear Analysis, Th., Meth.& App., 30(5), 2873-2882, 1997.

[6] G. Winter, G. Montero, L. Ferragut y R. Montenegro *"Adaptative strategies using standard and mixed finite elements for wind field adjustment"*, Solar Energy, 54, 1, 49-56, 1992.

[7] R.E. Bank, A.H. Sherman, A. Weiser, *"Refinement algorithms and data structures for regular local mesh refinement"*, in Scientific Computing IMACS, Amsterdam, North-Holland, 3-17, 1983.

[8] D.N. Arnold, A. Mukherjee, L. Pouly, *"Locally adapted tetrahedral meshes using bisection"*, SIAM J. Sci. Comput., 22(2), 431-448, 2000.

[9] M.C. Rivara, C. Levin, *"A 3-d refinement algorithm suitable for adaptive multigrid techniques"*, J. Comm. Appl. Numer. Meth., 8, 281-290, 1992.

[10] A. Plaza, G.F. Carey, *"Local refinement of simplicial grids based on the skeleton"*, Appl. Numer. Math., 32, 195-218, 2000.

[11] F. Bornemann, B. Erdmann, R. Kornhuber, *"Adaptive multilevel methods in three space dimensions"*, Int. J. Numer. Meth. Eng., 36, 3187-3203, 1993.

[12] A. Liu, B. Joe, *"Quality local refinement of tetrahedral meshes based on 8-subtetrahedron subdivision"*, Mathematics of Comput., 65(215), 1183-1200, 1996.

[13] R. Lohner, J.D. Baum, *"Adaptive h-refinement on 3D unstructured grids for transient problems"*, Int. J. Num. Meth. Fluids, 14, 1407-1419, 1992.

[14] R. Montenegro, G. Montero, J.M. Escobar, E. Rodriguez, J.M. Gonzalez-Yuste, *"Tetrahedral Mesh Generation for Environmental Problems over Complex Terrains"*, Lecture Notes in Computer Science, Springer Verlag, Berlin Heidelberg New York,335-344, 2002.