ELSEVIER

Advances in Engineering Software 34 (2003) 217–234

ADVANCES IN ENGINEERING SOFTWARE

www.elsevier.com/locate/advengsoft

An improved procedure for 2D unstructured Delaunay mesh generation☆

S. Secchi[a], L. Simoni[b,*]

[a]CNR, Ladseb, Padova Corso Stati Uniti, 4, 35127 Padova, Italy
[b]Dipartimento di Costruzioni e Trasporti, Università degli Studi di Padova Via Marzolo 9, 35131 Padova, Italy

Received 16 May 2002; accepted 1 November 2002

Abstract

This paper presents a procedure for the discretization of 2D domains using a Delaunay triangulation. Improvements over existing similar methods are introduced, proposing in particular a multi-constraint insertion algorithm, very effective in the presence of highly irregular domains, and the topological structure used together with its primitives. The method obtained requires limited input and can be applied to a wide class of domains. Quadrilateral subdivisions with a control of the aspect ratio of the generated elements can also be reached. Further it is suitable for evolutionary problems, which require continuous updating of the discretization. Presented applications and comparisons with other discretization methods demonstrate the effectiveness of the procedure.

Keywords: Triangulation; Quadrilateral conversion; Mesh generation; Finite elements

1. Introduction

One of the major tasks in finite element applications is discretization of the spatial domain. This is of paramount importance due to the effort required to prepare all the necessary input data and the fact that the solution possibly depends on it. Proper automatic procedures are usually a great help and improvements on these algorithms are continually being proposed. Structured/unstructured mesh generators in two and three dimensions, producing triangular/tetrahedral or quadrilateral/brick elements have recently been presented by Lee and Hobbs, Sarrate and Huerta, Radovitzky and Ortiz [1–3] for example. Books have been devoted to this topic [4] and a comprehensive handbook has also been published [5]. The last reference discusses the features of an efficient mesh generator for ensuring applicability to a wide range of cases together with the state of the art in this subject. To summarise the difficulties of this topic, the authors state in the preface that *grid generation is still something of an art, as well as a science*.

The importance of efficient mesh generators is greatly stressed in the presence of *h*-adaptive algorithms based on a posteriori local error estimators. These allow for new element size distributions, grid enrichment and derefinement based on previously obtained results and are particularly useful in the presence of evolutionary problems, e.g. in fracture mechanics, transport phenomena with phase changes located on moving fronts, strain localization and plasticity problems, etc. Unstructured meshes are frequently used in these cases to obtain a solution, which is certainly independent of the domain discretization.

Among the techniques leading to unstructured grids, the Delaunay–Voronoi methods offer the advantage of efficiency together with a sound mathematical basis and for these reasons they have been widely used, see Ref. [6] for an excellent survey. However, enhancement is still possible [7]. In this paper, an improved implementation of 2D Delaunay triangulation is presented. It pursues flexibility, reduction of user intervention, quality of the mesh and computational efficiency, independently of the complexity of the domain. Comparisons with other meshing algorithms, as far as complexity and CPU time are concerned, are shown, and help to prove the usefulness of the method.

The basic algorithm is the process by Rebay [8] for the triangulation of a 2D irregular domain using the Bowyer–Watson node insertion method (1981). An initial mesh is first described by using a minimum amount of information on the boundary of the domain (a set of vertices or a group of curves defining a closed domain), then it is selectively

☆ This paper is dedicated to Prof. Lorenzo Contri on the occasion of his 80th birthday.

* Corresponding author.
E-mail addresses: simoni@caronte.dic.unipd.it (L. Simoni), secchi@caronte.dic.unipd.it (S. Secchi).

0965-9978/03/$ - see front matter © 2003 Elsevier Science Ltd. All rights reserved.
doi:10.1016/S0965-9978(02)00131-X

refined controlling mesh grading through a user-defined node spacing function [9]. The introduction of a new point is attained by maintaining the information of all previously defined nodes and is performed in a sequential manner, dealing at the same time with the new element connections. It is well recognized that the efficiency of a Delaunay method relies on the efficiency in managing the element containing the point to be inserted and its neighbours [6]. To this end the capabilities of an object-oriented language are exploited and a modified edge-based data structure is proposed for storing mesh topology information and handling meshing operations. The usual topological clean-up tools are then applied to in order to control and enhance the quality of the subdivision.

The originality of the proposed mesh generator does not only rely on combining different existing procedures. Several enhancements and novelties have been introduced. First of all the data structure used, which improves the Quad-Edge presented by Guibas and Stolfi [10] for Voronoi tessellation and the Face-Based proposed by Weiler [11]. The first has been simplified for the dual diagram only and enriched with new primitives, for the second the high performances for adjacency relationships are retained, limiting memory requirements.

Additional refinements deal with a priori node grading along the boundaries and with the management of mesh and geometric singularities. The final discretization depends only on the initial description of the boundary (assigned nodes and spacing function) and there is absolutely no guarantee that with arbitrary input data a good quality mesh can be obtained. To improve this situation, the algorithm proposed in Section 3.6 corrects the boundary description when highly distorted elements originate from the assigned data, running from the initial steps of the generation. Good meshes are hence obtained by means of a unique process, without any correction phase, even in the presence of very irregular boundaries and geometric constraints. In this sense the subdivisions produced, although not optimised as finite element meshes, are certainly very appropriate from the geometric point of view and represent an optimal starting point for adaptive algorithms in numerical solutions. Problems which arise when geometric or mesh singularities are located near the boundary, generating high gradients of the spacing requirements, have also been solved. In these cases, the algorithm updates the grading along the boundary and improves the geometric description, maintaining the topology of the domain. The same procedure, which controls the grading along the boundary allows for quality control of the complete resulting grid.

Once the triangulation has been obtained, it can be converted into a quadrilateral mesh, which is sometimes preferred in finite element applications. The quality of the transformed mesh is ensured from the beginning, by optimising the aspect ratio for the generated quadrilaterals and by performing logical checks to choose between the different possibilities offered by the conversion.

From the programming point of view, the set of topological operators defined by Guibas and Stolfi has been enriched by the definition of new primitives and higher-level topological operators to perform the newly introduced tasks.

The procedure results in a 2D mesh generator which does not require orientation rules leading to the determination of the interior and exterior of the domain and is applicable to simply or multiply connected regions of any arbitrary shape, requiring the minimum amount of input data. The mesh generator can be used in adaptive refinement finite element applications as well as in the presence of material non-homogeneities. The latter are treated as different sub-bodies where different mesh sizes are allowed. Curved boundaries can be described in the usual way, either by interpolation or analytically. Quality checks will be presented to show that the procedure guarantees the creation of a good mesh for finite element applications and the computational efficiency will be proved both by theoretical considerations and practical tests.

The outline of the paper follows the logical order of the operations required to generate an unstructured mesh, but first of all the used data structure is described.

2. The topological structure

The starting point for the following developments are the techniques for managing generic oriented subdivisions, in particular the Quad-Edge structure proposed by Guibas and Stolfi [10]. This is popular in computer graphics and has been implemented in C^{++} along with its related pseudo-code by Lischinski [12]. However, it is not so common in grid generation [5]. Other edge-based structures, in the sense that information is contained in an edge-related arrangement, are common in computer graphics. The best known is probably the Winged-Edge introduced by Baumgart [13]. Structures of this type have been developed further [11,14] and are particularly useful for solid modelling in curved-surface environments and for non-manifold boundaries. A useful survey on this subject and related implementation can be found in Ref. [15]. Modifications to an existing structure or new proposals are in any case possible, together with a definition of the pertinent Euler functions, also depending on the structure's field of application. To create plane finite element subdivisions, the *TWEdge* data structure that simplifies the Quad-Edge and is similar to the Face-Edge suggested by Weiler [11] is proposed here. This structure preserves the edge algebra and all mathematical and topological properties stated for the Quad-Edge formulation as well as efficiency in handling the adjacency of the Face-Edge. In addition, it has been equipped with some new primitives directed to managing all the different meshing operations. As far as memory occupation is concerned, the following comparison between different structures can be made, accounting for

```
Tvertex{
   // Constructor (public)
            TVertex( double x, double y, double z);
   bool     IsEqual( double x, double y, double z, double toll);

   // Member Data (public)
   Tcoord*     pCoord;     // Vertex coordinates
   TEdge*      pEdge;      // Pointer to a connected edge
   TBoundary*  pBoundary;  // Pointer to the geometric entities defining the boundary
                           //     (=0 for internal points)
}
```

Box 1. *TVertex* object.

Euler–Poincaré relationships in the case of a large number of internal points and triangles.

Let n be the number of vertices, m the triangles and l the edges of the mesh. Whereas an explicit structure describing a triangular discretization needs $9n$ numbers (but information on adjacency is completely missing), in the edge-based data structures the amount of information ranges from $16n$ (directed-edge) to $30n$ (winged edge) numbers, as shown in Ref. [15]. Normally some more bits are necessary for successive operations. For instance, the Quad-Edge structure needs $27n + 36n$ bit memory occupation. The proposed *TWEdge* requires $4l + 4n + 3m \approx 22n$ numbers plus $24n$ bits (Boxes 1–4). As far as the data structures by Weiler [11] are concerned, they require from $24n$ numbers (modified Winged-Edge) to $33n$ (Face-Edge). When compared with structures maintaining adjacency information, the one proposed here requires less information, except from the Direct-Edge where, however, triangles are not explicitly represented.

The convenience and efficiency of the Edge type structures are sometimes questioned when dealing with mesh problems, on the basis that the natural entities involved are vertices and polygons (elements), not edges. A data structure with explicit reference to faces and vertices is hence suggested and the need for orientable surfaces is also sometimes relaxed [16]. In this case, a triangle-based structure requires $15n$ integer and float numbers in the presence of n vertices. However, an orientation of the neighbours of an element is necessary in any case, thus additional data are usually supplied in polygon-type structures. Moreover, information has to be saved on neighbouring edges of each polygon, consequently the entity *edge* is not completely expunged, but is always necessary when preparing a spatial subdivision, see Ref. [16] for more details. As a result, the polygon-type structure is certainly more natural in discretisation problems and involves a certain reduction in the data required. However, this does not imply clear computational superiority, especially when building, transforming or updating a mesh.

The choice of the data structure certainly results from a compromise between computational and storage efficiency. One of the advantages offered by the *TWEdge* structure is the drastic reduction in loops. The operations needed to construct an incremental Delaunay triangulation deal with creating, locating and deleting nodes, edges and triangles and require repetitive queries of neighbouring vertices, edges and polygons. These data are not searched for in arrays, but are already present in the *TWEdge* structure or

```
TEdge{
        //Constructor (public)
                    TEdge();

        // Attributes (public)
        TEdge*       Sym();
        TEdge*       O_Next();
        TEdge*       O_Prev();
        TEdge*       D_Next();
        TEdge*       D_Prev();
        TEdge*       L_Next();
        TEdge*       L_Prev();
        TEdge*       R_Next();
        TEdge*       R_Prev();

        TVertex*     Origin();
        TVertex*     Destination();

        TFace*       CwFace();
        TFace*       CcwFace();

        TVertex * pVertex;       // origin node (pointers to the global array)
        TFace*    pFace;         // left face following natural orientation
        char      Index;         // 0x01 : flag used to distinguish the position in TWEdge
                                 // 0x02 : ....
                                 // 0x06 :    bits used to represent the
                                 // 0x07 :    edge index of pFace associated
                                 // 0x08 :    to this edge
        . . .   // other public methods
}
```

Box 2. *TEdge* object.

```
TWEdge {
        // Constructor (public)
                TWEdge();
                TWEdge( TVertex* orgin, TVertex* dest );
        TEdge*   Edge();
        TEdge    pEdge[2];

        . . .    // other public methods
}
```

Box 3. *TWEdge* structure.

obtained by the pertinent Euler functions. Geometrical information is preserved; hence the queries are performed in a limited time, almost independently of the number of elements in the list. For this reason, the topological structure also allows for a rapid execution of other operations such as swapping of edges, mesh quality control and similar tasks. A significant example will be illustrated later. The *TWEdge* structure also has a fixed size, hence the memory allocation can be easily optimised.

Although not used in this paper, the *TWEdge* structure maintains the information for spatial operations (Boxes 1–4), in particular it is suitable for representing spatial surfaces and solids by means of triangles or polygonal surfaces of higher complexity and to deal with tetrahedral subdivisions. The usual topological transformations (translations, rotations, scaling) as well as more complex operations in space (such as sectioning of solids and their spatial visualisation) are also possible by adding a small amount of extra data.

The basic plane topological entities (Fig. 1) are vertices (*TVertex* object), edges (*TEdge* object) and faces (*TFace* object). Their union establishes the ordered complex *TWEdge* structure.

The structure elements must fulfil some requirements which make their application possible and some conditions ensuing from the topological consistency:

- each element is required to reach, by means of internal pointers, its corresponding neighbouring entity;
- each edge can be shared by no more than two faces (two-manifold);
- each edge must possess at least one face connected to it;
- each vertex must be connected to at least two edges;
- each face is defined by at least three vertices;
- there are no theoretical bounds to the number of edges connected to a vertex. This is a necessary requirement to obtain unstructured meshes.

Other information concerns the orientation. For each edge there is the possibility to define two orientations, the first, from the origin to the destination node (called *natural*) and the opposing one. For this reason in the *TWEdge* class two *TEdge* objects are present, with the same direction and opposite orientation.

In an object-oriented framework, the *TVertex* class manages the node. It contains (Box 1) a *TCoord* object, which controls the coordinates, and a pointer to one of the edges connected to the vertex. The set of edges joined to each vertex forms the node ring. This can be covered clockwise or counter-clockwise by using the edge operators presented in Appendix A.

The *TEdge* class contains and manages the information of each edge of the subdivision. The member data are (Box 2): *pVertex* pointer to the origin node, *pFace* pointer to the face located at the left of the edge when observing from *pVertex*, and *Index*, a short-type datum, i.e. a one byte char described in the following.

The three main entities (*TVertex*, *TWEdge*, *TFace*) of the topological structure refer to 1, 2 or 3 *TEdge* objects, respectively. In the first case, the pointer allows the object to know what and how many edges are linked to the vertex. Recognising whether the vertex belongs to the boundary of the domain without introducing new data is straightforward. The two pointers in *TWEdge* establish an oriented edge and, finally, the three pointers in *TFace* describe the edges of the triangular element of the subdivision.

The efficiency of the structure increases if the *TEdge* object is able to recognise itself within the other entities in which it is contained. This means that each *TEdge* object has to know its position within the arrays containing it, and present in *TWEdge* and *TFace*. The information contained in the *Index* datum is used for this purpose. By means of the first bit the object recognises its presence in the first or second position in the *pEdge* array of the *TWEdge* class. The following bits are flags of state (i.e. state of visualisation,

```
TFace {
    //Constructor (public)
                TFace();
    TEdge&      operator[] (int n);
    TVertex*    operator() (int n);
    virtual int NumEdges() {return 3;}
    bool        SetConnection ();
    bool        AdjustConnections (TEdge* poldConn[]);

    // Member Data (protected)
    TEdge**    pEdge;          // array of edges
}
```

Box 4. *TFace* object.

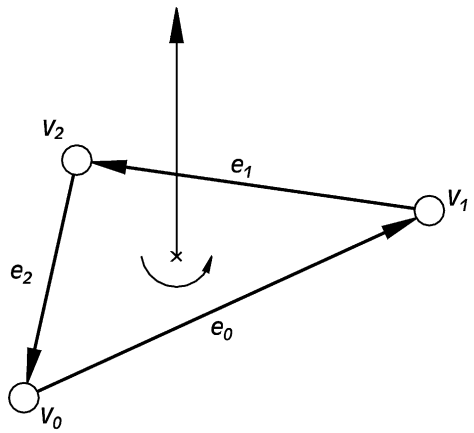


Fig. 1. Orientation of a face.

selection, etc.) and are not used in the following applications; the three final bits contain the digits 0, 1, …7, which define the position of the edge in the incidence array of the *pFace*. For a triangular mesh only two of these bits are necessary.

Depending on the way the edge operators are built, particularly for the function *Sym( )* to be operative, it is necessary that operations which create the *TEdge* objects be exclusively performed in pairs of *TWEdge* objects, by using dynamical memory allocation.

Box 2 lists the basic operators for the *TEdge* object. For instance *Sym( )* yields the pointer to the edge having an opposite direction to the one under consideration (Fig. 2). By using these operators the different entities necessary for meshing the domain are easily identified, e.g. it is possible (Fig. 2) to identify the vertex destination of edge *e*, which is represented by *e*- > *Sym*( )- > *Origin*( ), the vertex of the left triangle not shared by *e* is given by *e*- > *D_Prev*( )- > *Origin*( ). Appendix A gives a description of the main functions built for this class.

The *TWEdge* class collects two *TEdge*-type objects and produces the oriented edge (Fig. 3). An object *TWEdge* is associated to each side of the triangulation, hence the total memory occupation is $N\_edge \times 2 \times m$ bytes, *N_edge* being the total number of edges in the subdivision and *m* the memory occupation of a single *TEdge*, e.g. 9 bytes in a 32-bit system. Box 3 shows the class related to this structure.

The topological entities are completed by the structure *TFace*, which defines a face of the tessellation using *NumEdges*( ) edges suitably allocated in the computer memory. The functions *SetConnection*( ) and *AdjustConnections*( ) (Box 4) initialise and modify data related to the incidences of the analysed and neighbouring elements, respectively.

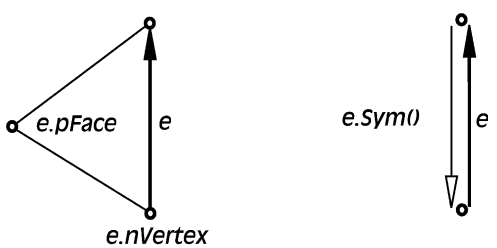


Fig. 2. Representation of a generic *TEdge*.

3. Basic features for the mesh generator

The proposed 2D mesh generator operates at the following steps:

- the domain to be discretised is described by the user by means of nodal points, analytical curves, plane polygons (simply or multiply connected). At nodal points and along each curved boundary the spacing function $\Im(x, y)$ has to be assigned, and this defines the maximum length of the sides of the final triangles [9]. The compatibility of the spacing function with all geometrical information is automatically checked and, if necessary, corrected (Section 3.6). Additional information can also be introduced as an option; for instance, concerning the presence of internal sub-regions of different materials, singularity zones where a finer subdivision is required, and geometrical parameters for checking the quality of the final mesh. Very limited input data are required in any case, but, more importantly, this step is the only one requiring the user's contribution. Further, input data can be obtained or updated on the basis of a previous analysis when dealing with evolutionary problems;
- a convex auxiliary domain is assumed, which contains the region to be meshed. For the sake of simplicity a triangle is assumed. No consequences arise from this choice;
- the auxiliary domain is triangularized by inserting boundary nodal points via the Bowyer–Watson algorithm. This result is attained by using the minimum number of subdivisions capable of representing the available information and produces the *boundary triangulation*. The procedure used guarantees that all

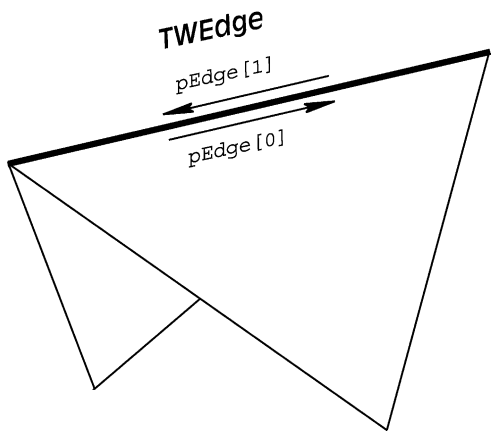


Fig. 3. Representation of a generic *TWEdge* in a 3D subdivision.

the generated triangles satisfy the Delaunay criterion, though, in general, not the spacing function requirements. All the created triangles not satisfying the spacing requirements are organised in a list ordered by increasing values of a quality measure of their shape (*bad-condition element list*). This measure is defined by the following;

- body consistency, i.e. representation of the complete boundary, is not guaranteed at this stage, since the Delaunay tessellation of a given set of points is a single construction, and as such is generally unable to represent the variety of boundary shapes. Once boundary triangulation has been obtained, the algorithm checks whether boundary integrity is preserved. If not, vertices are located along the missing boundaries by using the procedure illustrated in Section 3.1 and then triangulated. If it is the case, user assigned spacing function values are changed, until a closed polygon is obtained which represents the boundary [17]. This inspection is completed in the early stages of the triangulation when very few triangles have been generated, hence it does not require long execution time. In this task, the *TWEdge* structure proves very useful, for the orientation information contained and for the possibility to distinguish boundary edges from internal ones;
- all triangles external to the domain are deleted;
- a new vertex is inserted in the currently worst triangle to fit the requirements of the spacing function. The decision of where new points have to be introduced depends on the bad-condition triangle list. When this is empty, the procedure is stopped;
- the quality of the resulting mesh is improved through topological clean-up operations [18].

The listed operations are to some extent usual in these problems, however, here they are revised in order to indicate the novelties, together with some procedural remarks dependent on the data representation chosen.

3.1. Geometry description

Nodal points defining the geometry can also be interior to the domain and are assigned without any particular arrangement regarding their sequence. Empty internal subdomains or zones with different mechanical characteristics are defined by their boundaries with the only requirement that they are closed. The spacing function on the boundary vertices and internal points is directly assigned.

All previous data can be obtained from a prior finite element analysis, in particular the topological changes of the domain of analysis (e.g. fracture advancing), the position of possible singularities (points, lines, areas) and the spacing function values, requiring a finer mesh in zones with high gradients of field variables and/or their derivatives. In these cases only the definition of a suitable error measure is required to proceed with an *h*-adaptive refinement/derefinement based on spacing function updating. The spacing function is an assigned value at selected nodal points, ranging from a limited subset to all points. An interpolation (grading) along the boundaries is presented in Section 3.2. Only one parameter per vertex is hence needed. With this simple input, the concepts of point, line and area sources [19] can be recovered, even though the present approach seems simpler for an a posteriori refinement.

The grading of the values of the spacing function in the interior of the domain is obtained through piecewise linear interpolation at triangle level. However, the calculated or assigned spacing function values are automatically modified (Section 3.6) when inconsistencies are detected between different spacing requirements. Stricter requests are obviously preserved. For instance this is the case when a singularity approaches the boundary.

3.2. Point insertion in an existing triangle

The strategy proposed by Bowyer and Watson [20,21] is used here. First the triangle containing the new point P is found. This triangle is then eliminated and replaced with three new triangles obtained by connecting point P with the vertices of the old triangle. These are checked to be of the Delaunay type. When this does not hold, the *swapping* technique [22] is used, which involves changes in neighbouring triangles. The check is performed recursively on all triangles whose incidence has been changed and

```
TGeoMesh2D::Swap(TEdge* e)
{
   TEdge* pstart = e;
   do {
      e->pVertex = pnewNext->pVertex;   // Re-establishes the origin of the new edge
      e->Origin()->pEdge = e;           // ...updates the pointer to the origin
      (*e->pFace)[0] = e->O_Next()->Symm();
      (*e->pFace)[1] = e->O_Prev();
      (*e->pFace)[2] = e;
      for (int i=0; i<3; i++)
         (*e->pFace)[i]->SetFaceIndex(i);

      e = e->Symm();
   } while (pstart != e);

   Classifies(e->CcwFace()); // classifies the triangle (accepted or not..)
   Classifies(e->CwFace());  // and re-orders the ill-conditioned triangle list
}
```

Box 5. Swapping.

```
TEdge* Locate(const T2DPoint& x, TEdge *pStartingEdge)
{
   TEdge* e = pStartingEdge;
   int actualEdge = 0;
   while (actualEdge++ <= m_ArrayWEdge.size())
   {
      if (x == e->Origin() || x == e->Destination())
         return e;
      else if (RightOf(x, e))
         e = e->Sym();
      else if (!RightOf(x, e->O_Next()))
         e = e->m_pNext;
      else if (!RightOf(x, e->CcwEdge()->Sym()))
         e = e->CcwEdge()->Sym();
      else
         return e;
   }
   return 0;
}
```

Box 6. Determination of the pointer to the edge nearest to point P.

the bad-condition list is updated. The implementation of these operations accounting for the *TWEdge* information is shown in Box 5, demonstrating that no logical operators are required to find topological data. It is well recognised that the algorithm efficiency depends very much on how quickly the triangle containing P is found [6]. For this task, the capability of the *TEdge* structure and *Locate* function (Box 6) are of paramount importance and will be discussed.

3.3. Grading of nodes along the boundary

The position of new intermediate vertices along the boundary and the respective spacing function values are determined as follows.

Let $\Im_1$ and $\Im_2$ be the values of the spacing function at the extreme points of a straight boundary segment of length ℓ. These values must be modified if they are greater than ℓ, in which case they are assumed to be both equal to ℓ and no insertion is made. The length of the extreme sub-segments is assumed as $\Im_1$ and $\Im_2$ (Fig. 4).

Let r be the factor of a geometric progression and n the smallest positive integer less than x so that the following relationships hold:

$$r^0\Im_1 + r^1\Im_1 + r^2\Im_1 + \cdots + r^n\Im_1 = \Im_1 \sum_{i=0}^{n} r^i$$

$$= \Im_1 \frac{r^n \times r - 1}{r - 1} \tag{1}$$

$$\Im_2 = r^x \Im_1 \tag{2}$$

The problem is to find such values for r, x and n that Eq. (2) is satisfied together with the approximation of Eq. (1)

$$\ell = \Im_1 \frac{r^x \times r - 1}{r - 1} \tag{3}$$

Eqs. (2) and (3) yield

$$r = \frac{\ell - \Im_1}{\ell - \Im_2} \tag{4}$$

hence, from Eq. (2), x turn out to be

$$x = \frac{\log \dfrac{\Im_2}{\Im_1}}{\log \dfrac{\ell - \Im_1}{\ell - \Im_2}} \tag{5}$$

By using n instead of x, summation in Eq. (1) does not yield ℓ. Hence Eq. (3) is modified as

$$\ell_k = \Im_1 \frac{r^{\alpha(k+1)} - 1}{r - 1} \tag{6}$$

By assuming

$$\ell = \frac{r^{\alpha(n+1)} - 1}{r - 1} \Im_1 \tag{7}$$

parameter α can be obtained as

$$\alpha = \frac{\log\left(r \dfrac{\Im_2}{\Im_1}\right)}{(n+1)\log(r)} \tag{8}$$

Eq. (6) allows for a good node grading along the boundary, at the same time respecting the assigned values of $\Im$ at the limiting points. For the following computations, each point is assigned a spacing function value equal to the length of the preceding segment. Fig. 5 shows the results obtained for the discretization of a segment of length 100 for different values of $\Im_1$ and $\Im_2$. When these are equal, a uniform distribution of nodes is performed directly.

For curved boundaries the procedure just presented is used by assuming ℓ to be the length of the arc of the curve.

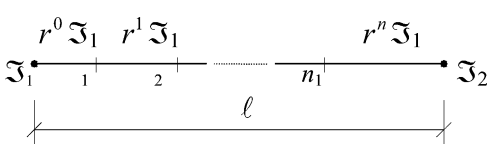


Fig. 4. Node insertion along a segment.

	$\Im_1$	$\Im_2$	x	n	α
1	1	10	24.16	24	1.0064
2	2	10	18.90	18	1.0473
3	5	10	12.82	12	1.0631

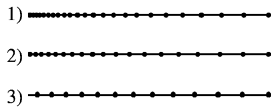


Fig. 5. Results for node insertion for a segment using different values of the spacing function at the ends.

3.4. Deletion of the external triangles

Once the boundary triangulation has been obtained, triangles connected to the vertices of the auxiliary triangle, located in concave zones and in internal cavities (if present) are eliminated. For this task, the *TWEdge* operator makes it possible to perform an oriented loop over all edges defining the boundary of the domain deleting all triangles lying on one side of the edge. This operation is very fast, due to the minimal number of triangles at this stage.

It is, however, advisable to maintain the information regarding the edges, especially in the presence of concave boundaries and internal holes. Elimination of a triangle means that it is eliminated from the pertinent list, while information about its edges is still maintained. Preserving edge-information accelerates the algorithm for insertion of a new vertex, based on the use of the *Locate* function. In fact, moving from a generic edge (*pStartingEdge*), this progressively approaches the insertion zone by using the topological information contained in *TWedge* and relevant primitives. The most advantageous path can cross the holes and concavities of the domain, which are described by means of edges only.

At this stage, the whole domain is covered by a consistent triangulation, but spacing requirements are fulfilled along the boundary only.

3.5. Insertion of internal nodes

Internal triangles are mostly distorted, with edge lengths greater than required by the discretization function, they are then stored in the bad-condition triangle list. The undistorted ones, of course, will not be modified and are classified as *accepted triangles*. This situation is improved through proper insertion of new vertices of the discretization. Several possibilities have been proposed in the literature, but in the authors' experience the best results are obtained by using the Rebay procedure (1991) and locating the point on the Voronoi tesselation. In a generic step, the triangle (*active triangle*) is operated, having at the same time the most unfavourable value of the quality measure parameter (defined in the following) and sharing an edge with an accepted triangle or having an edge lying on an external boundary. If the triangle has more than one edge of this type, the Voronoi segment of the shortest edge is considered. The position of the new point X depends on the following conditions (Fig. 6):

- it belongs to a circle touching through points P and Q and having a radius ρ smaller than that of the circumcircle of the active triangle;
- the radius of the circumcircle of the new triangle should be as close as possible—ideally equal to—the spacing function calculated in the centroid of the new triangle PQX. This value is unknown and can be approximated through the spacing function evaluated in a neighbouring point, for instance point M (Fig. 6), i.e. the intersection between the common edge (PQ) and Voronoi edge. This approximation usually has no

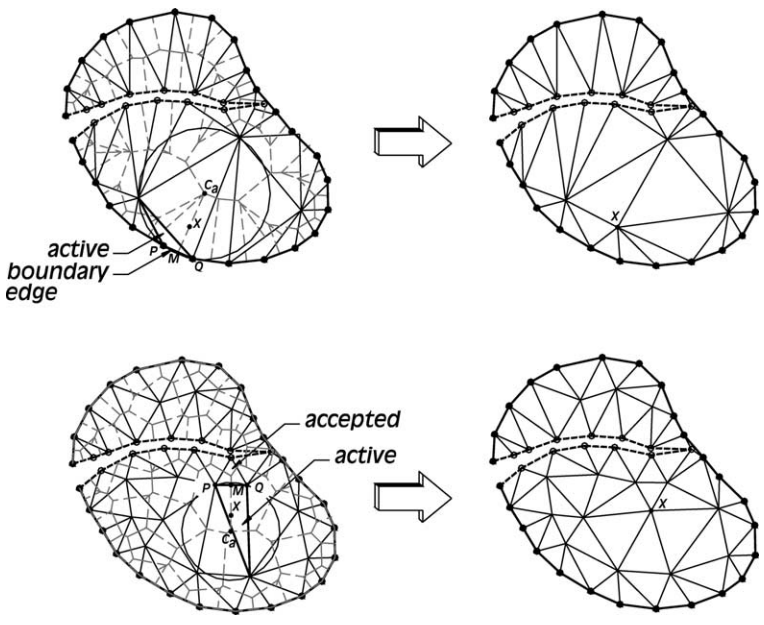


Fig. 6. Location of a new point on Voronoi segments.

particular consequences thanks to the regularity of the spacing function.

As a consequence, the distance d from X to M is

$$d = \hat{\rho} + \sqrt{\hat{\rho}^2 - p^2} \tag{9}$$

where

$$\hat{\rho} = \min\left[\max(\rho_M, p), \frac{p^2 + q^2}{2q} \right], \qquad p = \frac{1}{2}|PQ|, \qquad q = |C_a M|, \qquad \rho_M = \Im(M) \tag{10}$$

Point X will be connected to the vertices of the triangle containing it, as will be discussed in the following. However, before finally accepting the new point, a check is carried out to make sure that the length ℓ_i of all edges generated satisfies a condition of the type: $\ell_i > \beta\Im(x)$, β being a user-defined parameter, usually in the range 0.5–1. This avoids the generation of triangles which are too small and introduces an inferior limit to edge size. Further, it guarantees that the insertion procedure comes to a halt. If this check is not passed the triangle is accepted as it is, if the aspect ratio is good enough, otherwise a new point is inserted.

Once the optimal position of the new point X has been found, the following cases are possible:

- the point belongs to the active triangle and the insertion procedure of Section 3.2 is used;
- the point does not belong to the active triangle. This problem is solved by first finding the *TEdge* nearest the point (Box 6), then the triangle containing both the point and the *TEdge*. The insertion procedure is then applied to this triangle;
- the point lies outside the domain or very near to an existing vertex. No insertion is made and a new triangle is considered for insertion;
- the point lies near an existing edge. This edge and the triangles sharing it are deleted; point X is connected with the vertices of the deleted triangles. Four new triangles are in this way generated which are transformed into a Delaunay type, if necessary, by applying the swapping procedure.

Finally, the value $\Im_X$ in the new vertex is calculated by means of a bilinear interpolation over the triangle containing point X.

In all cases, it is needed to find the position of point X with respect to some edge. The *Locate* function proves to be efficient in this task because parameter *pStartingPoint* is an edge very near to point X and this query can be performed in a very limited time, almost independently of the number of elements.

In the cases in which the point is inserted, Delaunay type triangles are produced with a general gain in the spacing function. In all other cases the procedure runs simply by changing the order of the triangles considered.

The mesh quality measure α for ordering the generated triangles is assumed as

$$\alpha = \frac{\rho_G}{R} = \frac{\psi\Im(X_G)}{R} = \frac{f(X_G)}{R} \tag{11}$$

ψ being a suitable coefficient obtained on the basis of experience, ρ_G the radius of the circumcircle in the optimal final mesh, R the actual circumcircle radius, G the centroid of the actual triangle. Good results were always obtained when assuming $\psi = 0.7$.

3.6. Multi-constrained point insertion

The procedure so far described is unable to handle situations in which contrasting requirements of the spacing function arise, i.e. when the gradient of $\Im$ is too high or the value of the spacing function is not compatible with the geometry of the domain. This happens, for example, when a singularity point with small values of the spacing function is near the boundary, where $\Im$ presents higher values or when different sub-domains are too near with respect to the assigned spacing function. These cases are typical in fracture mechanics problems for instance, when the apex of the fracture approaches the boundary. Further, in the presence of irregular boundaries, holes and/or geometrical constraints, linear interpolation of the spacing function is not suitable and produces highly distorted triangles, inappropriate for use in a finite element analysis. Excluding user intervention, very cumbersome or practically impossible for multiply connected regions and moving discontinuities, the problem could be solved by changing the spacing function during node insertion. The multi-constrained insertion method allows for an a priori adjustment of function $\Im$ and involves a gain in execution time because all operations are performed on a very limited quantity of data.

Once the boundary mesh has been generated (Section 3.2) and body consistency is guaranteed, the triangles are checked to see if they will be able to fulfil the spacing and shape requirements at the same time. Let A be a point on the singularity area, A' its projection on the boundary Γ_1 and k the admissible ratio between the edges (Fig. 7). When

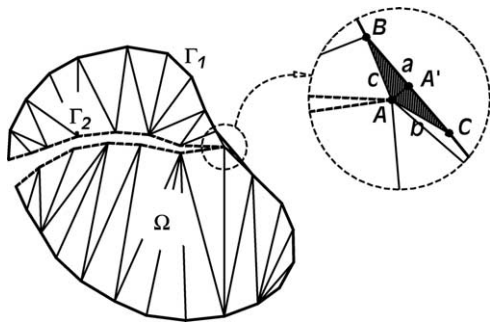


Fig. 7. Multi-constraint insertion geometry.

considering the triangle with a vertex in A and the others on Γ_1, if the inequality

$$\min\{\Im_A, \Im_{A'}\} > k\overline{AA'} \tag{12}$$

holds, there is certainly at least one triangle with a bad aspect ratio in the final mesh.

The algorithm searches for triangles which have vertices on two different boundaries or on a boundary and a singularity. Let B and C be the vertices belonging to the same boundary (Fig. 7). If the inequality

$$2k'\Omega_E/a < \max(\Im_B, \Im_C) \approx a \tag{13}$$

is satisfied where Ω_E is the area of the element, a the length of the boundary edge, $\Im_B$, $\Im_C$ the spacing function evaluated at points B and C and k' a user-defined parameter (we assume $k' = 1.5$), insertion of a new node in this triangle only results in a worsening of its aspect ratio. Improvements can only be obtained by locating new nodes along the boundary. Nodes A, B and C (Fig. 7) are assigned the same spacing function value $\Im = 2\Omega_E/a$ and a new node is located on the boundary between B and C. In the presence of curved boundaries, the procedure manages the curve (which is known or interpolated) directly, without rough approximations of the real geometry as can be obtained by operating simply on the edge of the triangle. Vertex B or C, by using pointer *pBoundary* * present in *TVertex* (Box 1), requires the boundary to insert the new node. Its position is determined by a projection (orthogonal to the curve) of a point assumed in the middle of the edge $B-C$, or alternatively, locating the point along the curve at the middle of the curvilinear abscissa. The function *Locate*( ) (Box 6), operates in the usual way even if the new point is external to the previous boundary mesh. This procedure does not interact with boundary recovery operations, which have been performed in a previous step, but improves the representation of the real domain, in the presence of curved boundaries.

Two orders of problems arise in this approach. First, it is necessary to decide how many nodes must be introduced between B and C and where they must be located to fulfil the new spacing function requirements. Second, spacing function gradients must be reduced along the boundary external to $B-C$. In both cases new nodes are inserted between two adjacent points, until all spacing requirements are satisfied (Box 7).

Remark. The solution to this problem requires, for the topological structure, a link with the geometrical objects through the otherwise unnecessary pointer *pBoundary*.

3.7. Updating the triangulation

Once the new point has been located and the triangulation performed, accepted triangles are removed and the bad-condition element list is updated on the basis of parameter α. The procedure thus converges towards a Delaunay triangulation which fulfils the necessary quality requirements at the same time. This is achieved when the list of non-accepted triangles is empty and is obtained by increasing the total number of nodes, edges and triangles, as can be seen in Fig. 8. Topological clean-up operations are eventually performed to improve the regularity of the obtained mesh.

4. Generation of quadrilateral elements

Quadrilateral are usually preferred to triangular meshes because of their superior performance. For this reason, once a triangular mesh has been generated, the procedure makes it possible to convert it into a quadrilateral one, still taking into consideration the previously assigned discretization function.

Several methods are present in the literature for generating a quadrilateral mesh starting from a triangular one [23–26]. The proposed procedure, which is aimed at minimizing the number of isolated triangles and checking the regularity of resulting quadrilaterals, exploits the versatility of the topological structure and the speed of its primitives in the search for the optimal couple of triangles to be converted.

4.1. The regularity index

A regularity index is assigned to each element, which qualifies it for use as a finite element. For triangular elements, ratios between minimum and maximum edge length or internal angles represent obvious regularity measures, with the maximum associated to unity value. When dealing with quadrilateral finite elements, the regularity index is associated with two parameters, i.e. ratios

```
...
// distance between two adjacents points s, s-1 on boundary r
   dist = pb[r]->ArrayPoint[s-1]->Distance(pb[r]->ArrayPoint[s]);
// distance between previous adjacents points s-1, s-2 on boundary 'r'
   if (s>1) dist_prev = pb[r]->ArrayPoint[s-2]->Distance(pb[r]->ArrayPoint[s-1]);
   else dist_prev  = dist;
// ratio between two adjacents segments
   ratio = dist/dist_prev;
theoreticalratio = ::Min(pb[r]->ArrayPoint[s]->SpacingFunction,
pb[r]->ArrayPoint[s-1]->SpacingFunction) /dist;

if( ratio<0.5 || ratio>2 || theoreticalratio<0.5 )   {
  //  insert a new point along the segment
        InsertPoint(pb[r]->ArrayPoint[s-1], pb[r]->ArrayPoint[s], 0.5); }
```

Box 7. Insertion of new points along the boundary.

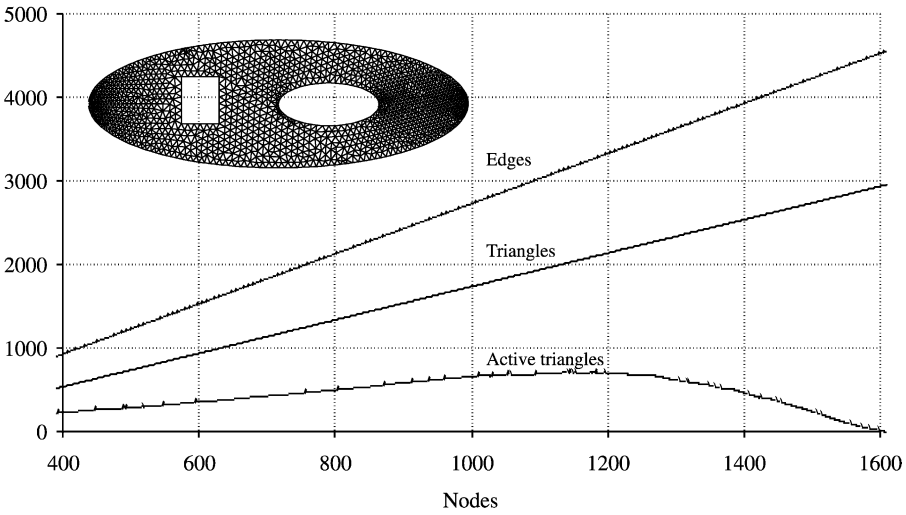


Fig. 8. Number of Edges, Triangles and Active Triangle vs. no. of Nodes in the ellipse case.

R_1, between minimum and maximum edge length, and R_2, between minimum and maximum diagonal length. The ensuing regularity index is assumed as

$$R = w_1 R_1 + w_2 R_2 \tag{14}$$

where w_1, w_2 are suitable user-defined weights, the sum of which must be equal to one, and are usually assumed as 0.5.

4.2. Mesh generation

A very simple technique consists in *splitting* the generic triangle into three quadrilaterals by using the Voronoi tessellation. The results are however quite poor in terms of mesh regularity.

Better results are obtained through the following generation process (Fig. 9):

1. identify a moving boundary which separates the triangle-only from the area with quadrilaterals. Initially, this boundary coincides with the external boundary;
2. consider the triangles with two edges on the moving boundary. These triangles, together with those sharing their single internal edge, are joined to obtain a quadrilateral element. The orientation properties of the Edge structure make this task very easy;

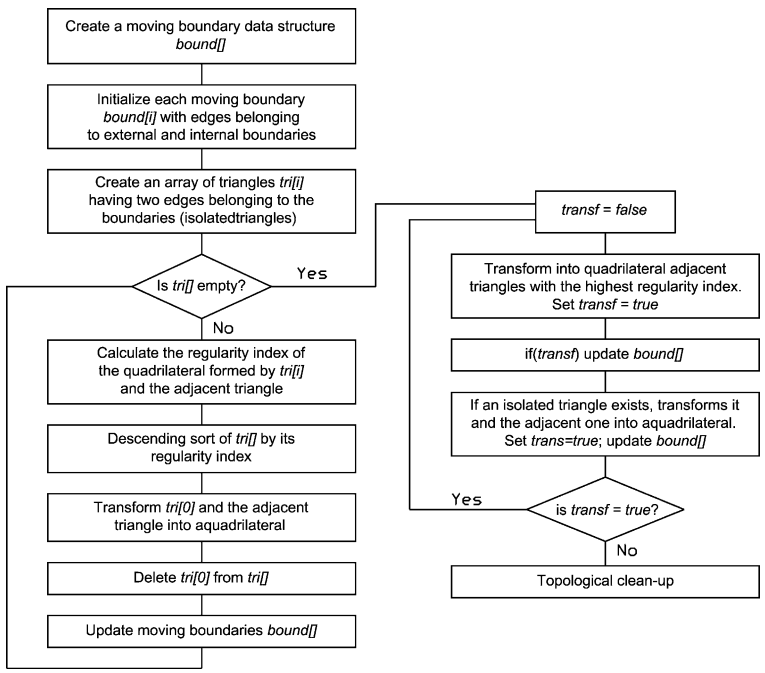


Fig. 9. Flow chart of the quadrilateral conversion procedure.

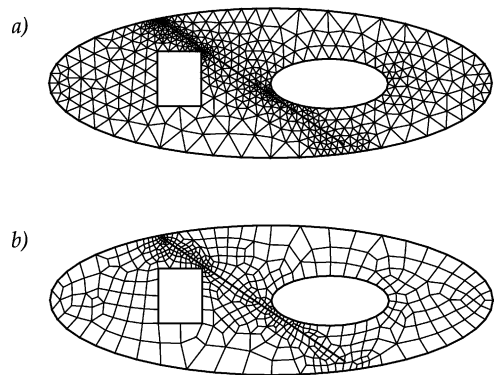


Fig. 10. Quadrilateral mesh (b) obtained from a triangular mesh (a).

3. transform into quadrilaterals all the adjacent triangles sharing at least one edge with a quadrilateral element;
4. find the pair of triangles on the moving boundary which form the quadrilateral with the best regularity index. Continue with step 2 until no more pairs of triangles are present on the boundary. As the transformed triangles are dropped from the relevant list, the front of the triangular mesh moves to the inner part of the domain. This is done by simply updating the list of Edges constituting the moving boundary;
5. once the triangular front is in the updated position, steps 2–4 are repeated. The procedure is stopped when the moving boundary has collapsed to unpaired triangles, to a single one or to a point;
6. topological clean-up procedures are used to regularise and improve the spatial discretization [18]. At the end of this procedure there may remain isolated triangular elements.

However, to limit the number of these triangles, logical checks have been introduced which act at the moment of choosing the pairs of triangles to be transformed into quadrangles (Fig. 9). The resulting procedure has proved to be fast and produces meshes of appreciable quality (Fig. 10). Applications are presented in the following.

By using the Swap( ) procedure and properly dropping internal edges, two isolated, but not too distant, triangles can be moved closer and converted into a quadrilateral element. Eventually the remaining triangles are eliminated by means of the splitting procedure.

5. Computational performances

5.1. Computational complexity

The incremental Delaunay triangulations are known to have an asymptotic complexity $\Theta(N^2)$ for triangulate N points. Location of the point to be inserted is the most expensive operation, requiring at worst, O(N) time per point. However, as previously noted, exploiting the properties of the *Locate* function, this time can be greatly reduced so that it can be assumed as nearly independent of N. The complexity hence becomes O(N).

As far as the present implementation is concerned, apart from the initial boundary triangulation and multi-constrained point insertion, which are performed on a limited number of elements, the complexity of the procedure is conditioned by the management of the bad-condition triangle list. This usually contains only n elements, with n somewhat smaller than the total number of elements. During the initial steps, all elements could be assumed as active (i.e. bad-conditioned) at worst, but in this case n is small. With proceeding triangulation the ratio of active over created elements continuously changes, always remaining unknown, but decreasing with increasing N (Fig. 8). The insertion of a new triangle takes place in an already ordered list, hence bisection is used for reordering it, which requires O(log n) operations for each insertion. The total cost cannot be determined a priori. In all applications made, a posteriori evaluations demonstrate a complexity ranging from O($N^{1.5}$) to O(N log N). The bad-condition element list can also be dropped, with a small loss in quality of the elements produced in the case of coarse meshes. In this case, the complexity of the overall procedure is O(N). This possibility will be exploited in a following application.

Other authors present algorithms of time complexity of O(N log N) [6], O($N^{1.5}$) [1], whereas the original algorithm by Guibas and Stolfi [10] requires a total time of $\Theta(N^2)$ in the worst case. Therefore, the proposed procedure has at least the same efficiency as others, but becomes better if some options are not activated. Similar conclusions can be drawn for the converter to quadrilateral mesh.

5.2. Quality assessment

The quality of the mesh is judged by means of the global quality index $Q = \sum_i A_i e_i / A$, where A_i is the area of the ith element, A the area of the whole domain, e a measure of the distortion of the element (the ratio between the maximum and minimum edge of the generic element is adopted). Hence Q is greater than 1.0 and tends to the unity for meshes composed of equilateral triangles or squares. The deviation from an equilateral triangle, i.e. the internal angle γ which maximizes the difference $|\gamma_i - 60|$, $i = 1, 2, 3$, is also assumed for quality assessment. For quadrilaterals, the regularity index of Eq. (14) is used with $w_1 = w_2 = 0.5$.

5.3. Numerical comparisons

The Delaunay mesh generation procedure presented has been implemented on a PC Pentium III 1200 MHz, 128 MB ram. In order to make a comparison, the authors have collected some data on time computation as presented by different workers in the field of a priori mesh generation, although of the opinion that this type of assessment is rather difficult, depending not only on hardware availability, but also on programming choices. The same environment as

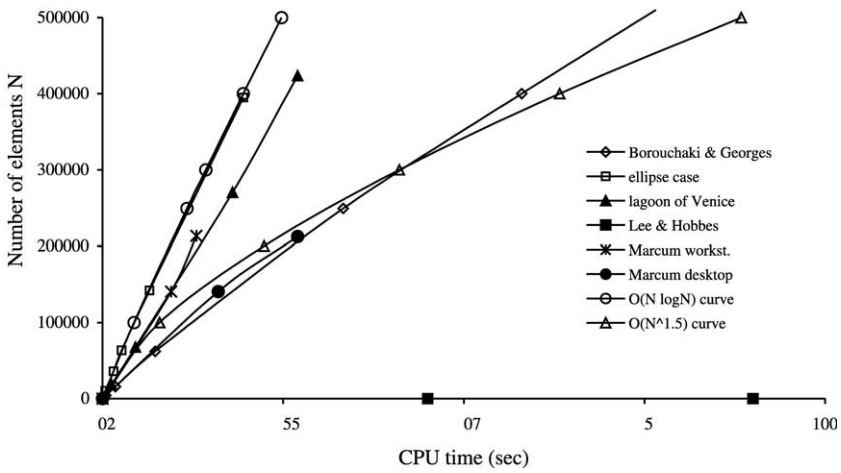


Fig. 11. Comparison of CPU time vs. generated elements for different triangular mesh generators.

used by Marcum [27] is adopted, controlling CPU times for I/O and generation of grid quality data. A discretized node grid is the input, strictly necessary to define the domain. The output includes grid coordinate, connectivity and quality data files. The results are shown in Fig. 11. As can be seen, the proposed algorithm implemented on a desktop computer gives the better performances than shown by Marcum [27] for a workstation, and Borouchaki and George [7] (quite old workstation). Computation times with respect to desktop implementation by Marcum [27] are halved.

6. Applications

Three applications of the proposed procedure are presented. The first test deals with the triangulation of an

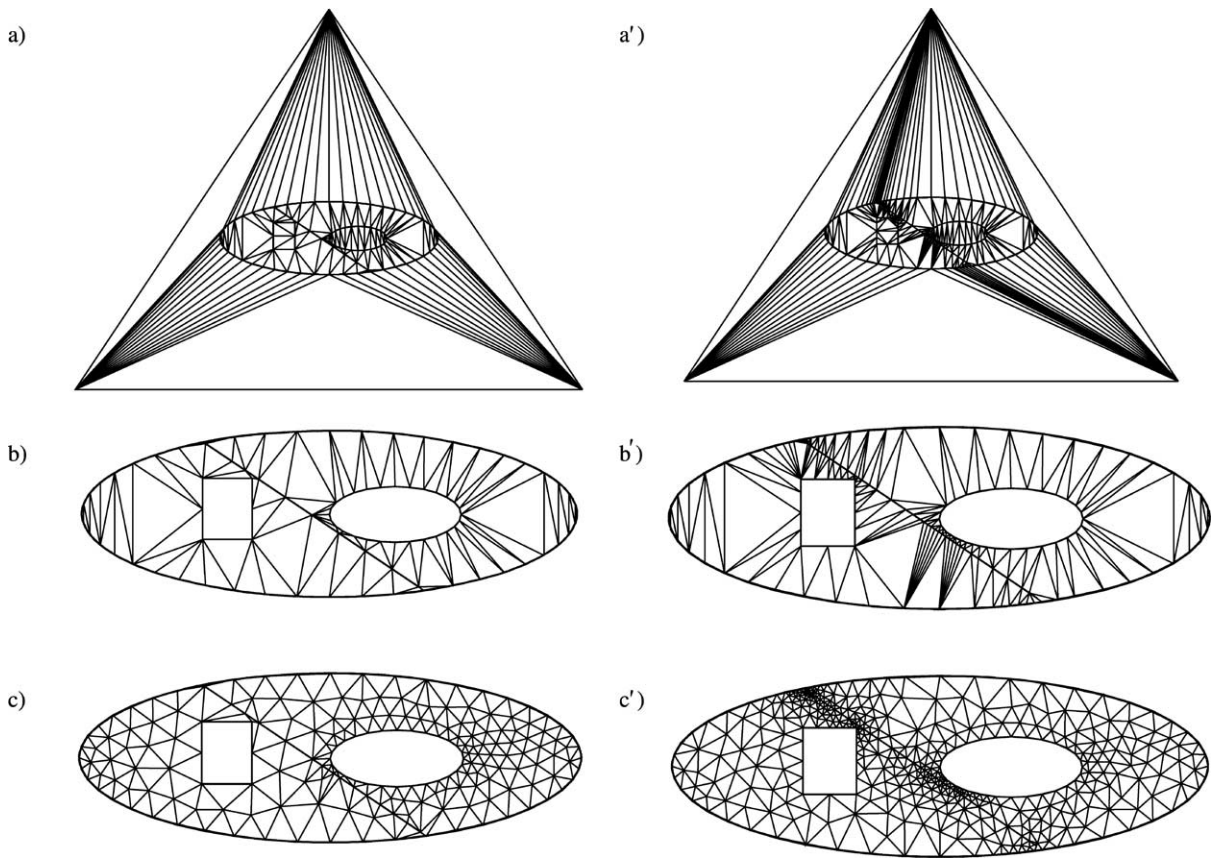


Fig. 12. Ellipse test case without applying the multi-Constraint insertion algorithm (a, b, c) and using the algorithm (a′, b′, c′).

Table 1
Results for the 'ellipse' case (for quadrilateral meshes required CPU time is given for conversion of the previous triangulation)

Case	1	2	3	4
Number of nodes, N	5386	18,360	32,178	71,832
Number of edges, E	15,717	54,255	95,433	213,845
Number of triangles, T	10,330	35,894	63,254	142,012
CPU time, t (s)	0.38	1.53	2.64	6.50
Q	1.09	1.06	1.04	1.02
T/t	27,184	23,460	23,960	21,848
Number of badly shaped elements ($e > 2.0$)	1	0	0	0
Number of quadrilateral	5094	17,722	31,278	70,196
Number of isolated triangles	7	20	35	75
CPU time (s)	0.46	1.69	3.07	7.47
T/t	11,074	10,486	10,188	9137

ellipsoidal domain with two internal cavities and an inclined discontinuity line. The initial boundary spacing is assumed constant along all the entities defining the boundaries and discontinuity: for the internal ellipse the spacing value is half that assigned to the other parts. The intermediate steps shown in Fig. 12(a) and (b) present the effects of the procedure of boundary nodes insertion and external triangle deletion (Sections 3.1–3.3), respectively. As can be seen in Fig. 12(c), along the discontinuity line and in particular at the upper-left end, elements are present with a distortion e greater than 10. 179 Nodes, 458 Edges and 278 Triangles have been generated and the mesh quality parameter is $Q = 1.41$.

In Fig. 12(a)–(c) the effects of the proposed multi-constrained point insertion procedure are presented. The result is a larger number of geometrical entities (311 Nodes, 827 Edges, 515 Triangles) with a major concentration in the zones where the spacing function gradient is higher. Improvements have been achieved for the mesh quality index, which now equals 1.21, and maximum element distortion, now equal to 2.25.

Table 1 summarises the information for the meshes (triangular and quadrilateral), the execution time and quality measures for different initial values of the spacing function. It can be seen that increasing the number of triangles eventually produces a mesh which comprises almost equilateral triangles ($Q \rightarrow 1$). Further, the generation speed (the throughput) decreases with increasing elements, both for triangular and quadrilateral meshes. Careful numerical analysis shows a complexity of $O(N \log N)$. Fig. 13 shows a histogram for the angle γ, showing that 90% of triangles are in the range 50–70° and 96% are in the range 40–80°.

The second example concerns the Venice lagoon (Fig. 14(a)), which in recent years has been the subject of several mathematical and physical models, especially for the analysis of environmental problems (tides, transport of contaminants, subsidence, etc.). Fig. 14(b) shows the schematic geometry of the multi-connected domain to which the discretization procedure is applied. Uniform spacing values are assigned to all boundaries. This case can also be assumed as a severe test for boundary integrity representation.

Without the multi-constraint insertion algorithm the practically homogeneous mesh of Fig. 14(c)–(e) is obtained. Instead, by using this procedure, discretization

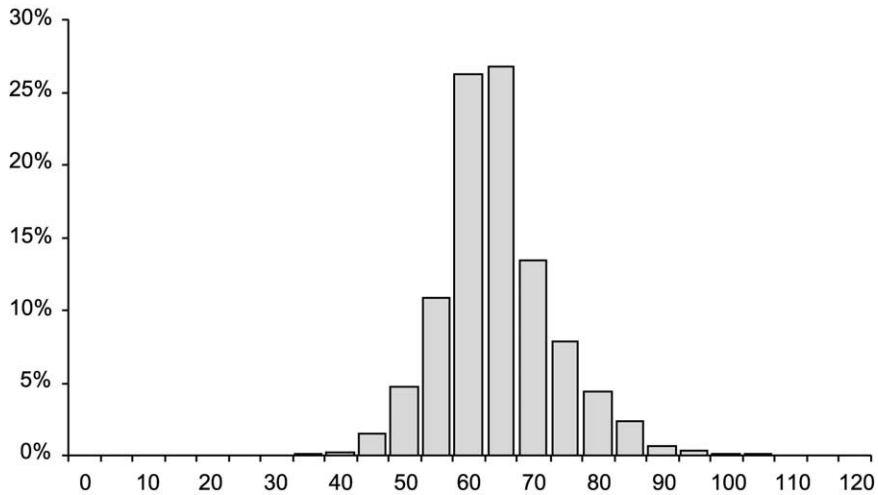


Fig. 13. Histogram for the maximum or minimum internal angle of the triangles.

Fig. 14. Venice Lagoon example.

is refined in the areas presenting geometrical details of *small* dimensions or too near contours (e.g. inlets). This clearly results from Fig. 14(d)–(f) where the transformed quadrilateral mesh is presented.

Table 2 summarizes some information concerning the generation procedure. It is noteworthy that the speed of generation is nearly constant owing to the fact that the sorting algorithm for reordering the bad-condition element list is not used. In the presence of fine meshes, in fact, any element of this list can be processed without loosing quality. In this circumstance, the operation count O(N) is confirmed. Similar behaviour is shown by the converter to quadrilateral elements.

Table 2
Results for the Venice lagoon case

Case	1	2	3	4
Number of nodes, N	10,652	36,495	141,199	643,480
Number of edges, E	28,995	104,135	411,716	219,593
Number of triangles, T	18,172	67,448	270,366	423,753
CPU time, t (s)	1.25	4.55	17.73	26.07
Q	1.08	1.04	1.02	1.01
T/t	14,512	14,816	15,250	16,256
Number of badly shaped elements ($e > 2.0$)	35	0	0	0
Number of quadrilateral	8864	32,742	132,532	209,779
Number of isolated triangles	34	7	12	51
CPU time (s)	0.85	3.8	14.70	22.02
T/t	10,428	8616	9015	9526

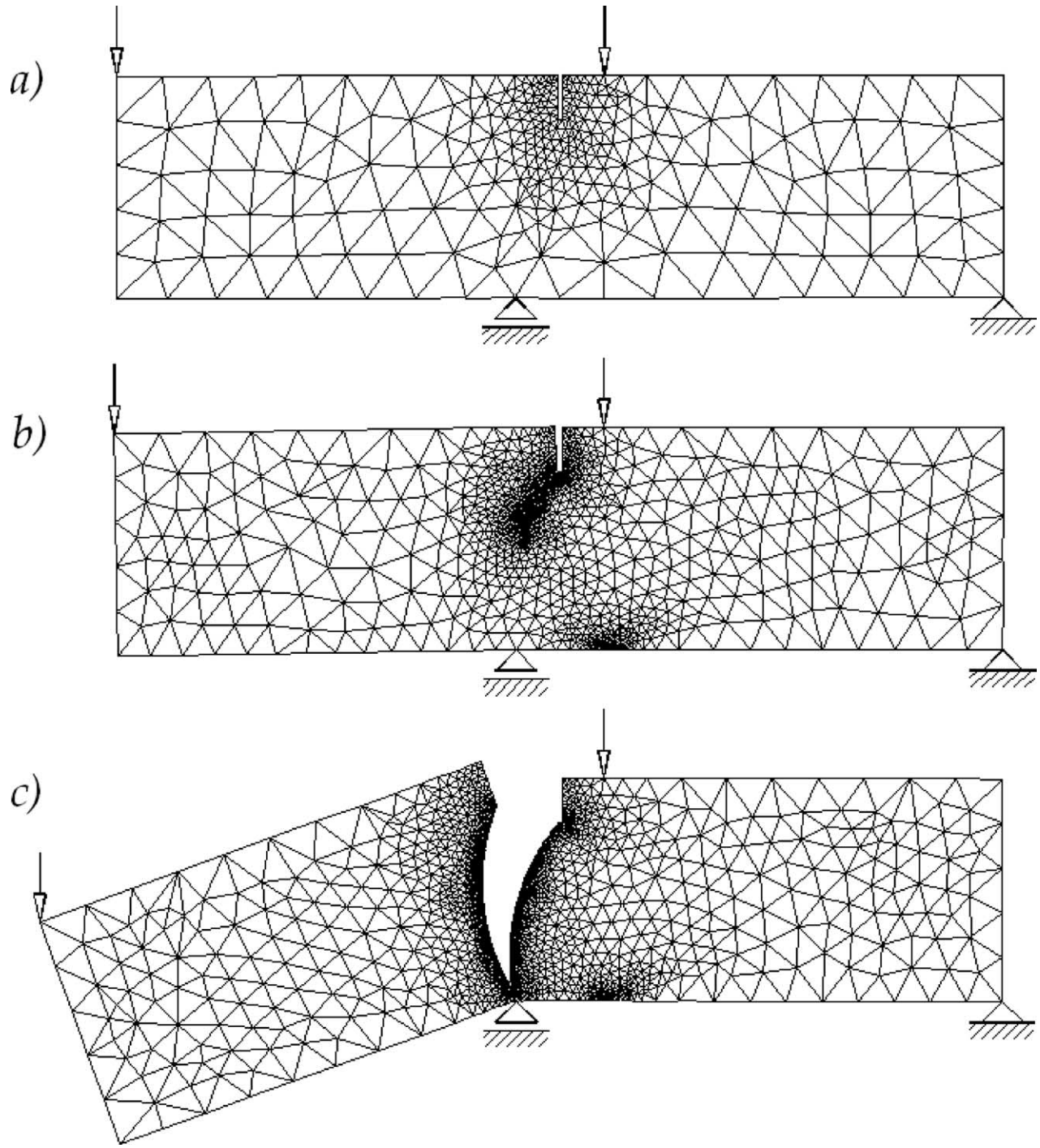


Fig. 15. Adaptive mesh generation in the evolutionary problem of crack propagation in a cohesive material.

The last example regards the evolutionary problem of crack propagation in a cohesive material, in particular the four point shear test proposed by Bocca et al. [28], which is now a benchmark problem in fracture mechanics. The boundary of the domain is initially assigned with a uniform spacing value independently of the specific dimensions of the beam and notch. The proposed procedure recognizes the singularity of the notch and automatically produces the triangulation shown in Fig. 15(a). With increased load, the procedure updates the domain, reassigns the spacing values and creates a new discretization with mesh refinements/derefinements dependent on the adaptive criterion used [29] and crack tip(s) movement. The other parts of Fig. 15 respectively show:

(b) the deformed mesh in an intermediate step, when two cracks have been enucleated near the left support;
(c) the amplified deformed mesh in step immediately preceeding the failure;

(d) the amplified deformed mesh with cohesive forces in steps immediately preceding the failure.

In this application, the limited amount of user required data is remarkable, i.e. the definition of the initial boundaries and a trial distribution of the spacing function.

7. Conclusions

The paper presents a powerful procedure, based on Delaunay triangulation, for discretization of 2D simply or multiply connected domains. The effectiveness has been proved with applications to both academic and real problems producing appreciable results in terms of execution time and discretization quality for use in finite elements codes. The latter aspect is validated by the fracture mechanics application.

The efficiency of the procedure mainly relies on the multi-constraint insertion algorithm, which modifies the user introduced spacing function and improves the shape of the elements, and on the proposed topological structure, with the related primitives, which, apart from its flexibility, allows for an optimal management of computer memory.

Acknowledgements

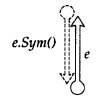
This research was partially supported by the Italian Ministry of University and Scientific and Technological Research (Grant MURST ex 60%) and (**Cofinanziamento MURST MM08161945_003).**

Appendix A

List of the principal functions operating in the class of Edge. Symbol % represents the modulus operator.

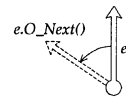
```
TEdge* TEdge::Sym()
{ return (Index & 1) ? this-1 : this+1; }
```

Returns the edge pointer from the destination to the origin.



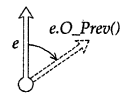

```
TEdge* TEdge::O_Next()
{ return L_Prev ->Sym(); }
```

Returns the pointer to the following edge present in a counterclockwise direction starting from the origin of edge *e*.




```
TEdge* TEdge::O_Prev()
{ return Sym()->L_Next(); }
```

Returns the pointer to the following edge present in a clockwise direction starting from the origin of edge *e*.

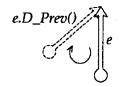



```
TEdge* TEdge::D_Next()
{ return Sym()->L_Prev; }
```

Returns the pointer to the edge of the triangle on the right. The edge is obtained by rotating the initial edge in a counterclockwise direction.

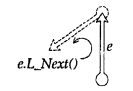

```
TEdge* TEdge::D_Prev()
{ return L_Next->Sym(); }
```

Returns the pointer to edge of the triangle on the left. The edge has the same destination point as *e*.



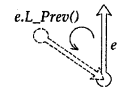

```
TEdge* TEdge::L_Next()
{ return pFace[ (IndexOnFace()+1) %3 ]; }
```

Returns the pointer to the edge of the triangle on the left. The edge is obtained by rotating the initial edge in a counterclockwise direction.



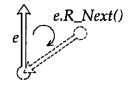

```
TEdge* TEdge::L_Prev()
{ return pFace[ (IndexOnFace()+2) %3 ]; }
```

Returns the pointer to the edge of the triangle on the left. The returned edge has a destination point coincident with the origin of *e*.



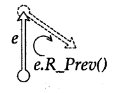

```
TEdge* TEdge::R_Next()
{ return Sym()->L_Next()->Sym(); }
```

Returns the pointer to the edge of the triangle on the right. The returned edge has the destination point coincident with the origin of *e*.




```
TEdge* TEdge::R_Prev()
{ return Sym()->L_Prev()->Sym(); }
```

Returns the pointer to the edge of the triangle on the right. The returned edge has its origin point coincident with the destination point of *e*.




```
int TEdge::IndexOnFace()
{ return (Index & 0xC0)/0x40;}
```

Returns the indexes 0, 1 or 2 representing the position of the edge, in the *pEdge* array contained in *pFace*.

```
void TEdge::SetIndexOnFace(int i)
{  Index &= 0x3F; Index |= (i<<6); }
```

Stores the numbers 0, 1 or 2 in the short-type datum *Index* using its last two bits. The stored number represents the position of the present edge in the *pEdge* array contained in *pFace*.

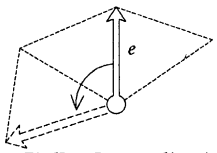
```
TFace* Edge::CcwFace();
{ return pFace;}
```

Returns object *TFace* present on the left of the edge.

```
TFace* Edge::CwFace();
{ return Sym()->pFace;}
```

Returns object *TFace* present on the right of the edge.

```
TEdge*  TEdge::FindLastConnected(bool ccw)
{    TEdge* pedge;
if (ccw)  // counterclockwise
{  pedge = pNext;
   while ( pedge->Ccw_Face() && pedge != this )
         { pedge = pedge->O_Next(); }
}
else  // clockwise
{  pedge = pedge->O_Prev();
   while ( pedge->CwFace() && pedge != this)
         {pedge = pedge->O_Prev(); }
   }
   return pedge != this ? pedge: 0;
}
```



e.FindLastConnected(true)

Scans the ring and returns the first edge (if any) of the boundary of the domain.

References

[1] Lee CK, Hobbs RE. Automatic adaptive finite element mesh generation over arbitrary two-dimensional domain using advancing front technique. Comput Struct 1999;71:9–34.

[2] Sarrate J, Huerta A. Efficient unstructured quadrilateral mesh generation. Int J Numer Meth Engng 2000;49:1327–50.

[3] Radovitzky R, Ortiz M. Tetrahedral mesh generation based on node insertion in crystal lattice arrangements and advancing-front-Delaunay. Comput Meth Appl Mech Engng 2000;187:543–69.

[4] Carey F. Computational grids: generations, adaptation, and solution strategies. London: Taylor & Francis; 1997.

[5] Thompson JF, Soni BK, Weatherill NP. Handbook of grid generation. Boca Raton: CRC Press; 1999.

[6] Baker TJ. Delaunay–Voronoi methods. In: Thompson JF, Soni BK, Weatherill NP, editors. Handbook of grid generation. Boca Raton: CRC Press; 1999.

[7] Borouchaki H, George PL. Aspects of 2-D Delaunay mesh generation. Int J Numer Meth Engng 1997;40:1957–75.

[8] Rebay S. Efficient unstructured mesh generation by means of Delaunay triangulation and Bowyer–Watson algorithm. J Comput Phys 1993;106:125–38.

[9] Frey WH. Selective refinement: a new strategy for automatic node placement in graded triangular meshes. Int J Numer Meth Engng 1987;24:2183–200.

[10] Guibas LJ, Stolfi J. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. ACM Trans Graph 1985;4:74–123.

[11] Weiler K. Edge-based data structures for solid modeling in curved-surface environments. IEEE Comput Graph Appl 1985;5(1).

[12] Lischinski D. Incremental Delaunay triangulation. In: Heckbert P, editor. Graphic gems IV. New York: Academic Press; 1994.

[13] Baumgart B. Winged-edge polyhedron representation for computer vision. AFIPS Proc 1975;44:589–96.

[14] Weiler K. The radial-edge structure: a topological representation for non-manifold geometric boundary representation. In: Wozny MJ, McLaughlin HW, Encarnacao JL, editors. Geometric modeling for CAD applications. New York: North Holland; 1988. p. 3–36.

[15] Campagna S, Karbacher S. Polygon meshes. In: Girod B, Greiner G, Niemann N, editors. Principles of 3D image analysis and synthesis. Boston: Kluwer Academic Publishers; 2000. p. 142–52.

[16] Zorin D. Subdivision for modelling and animation, SIGGRAPH; 2000, http://mrl.nyu.edu/ ~ dzorin/sig99/.

[17] Schroeder WJ, Sheppard MS. Geometry-based fully automatic mesh generation and the Delaunay triangulation. Int J Numer Meth Engng 1988;26:2503–15.

[18] Canann SA, Muthukrishnan SN, Phillips RK. Topological refinement procedures for triangular finite element meshes. Engng Comput 1996; 12:243–55.

[19] Weatherill NP, Marchant MJ, Hassan O, Marcum DL. Grid adaptation using a distribution of sources applied to inviscid compressable flow simulations. Int J Numer Meth Fluids 1994;19:739.

[20] Bowyer A. Computing Dirchelet tessellations. Comput J 1981;24(2): 162–6.

[21] Watson DF. Computing the *n*-dimensional Delaunay tessellations with application to Voronoi polytopes. Comput J 1981;24(2):167–72.

[22] Lawson CL. In: Rice JR, editor. Software for C 1 surface interpolation. Mathematical Software III, New York: Academic Press; 1977. p. 161–94.

[23] Baehmann PL, Wittchen SL, Shephard MS, Grice KR, Yerry MA. Robust geometrically-based automatic two dimensional mesh generation. Int J Numer Meth Engng 1987;24:1043–78.

[24] Rank E, Schweingruber M, Sommer M. Adaptive mesh generation and transformation of triangular to quadrilateral meshes. Commun Appl Numer Meth 1992;9:121–9.

[25] Lee CK, Lo SH. A new scheme for the generation of a graded quadrilateral mesh. Comput Struct 1994;52:847–57.

[26] Owen S, Staten JML, Canann SA, Saigal S. Q-Morph: an indirect approach to advancing front quad meshing. Int J Numer Meth Engng 1999;44:1317–40.

[27] Marcum DL. Unstructured grid generation using automatic point insertion and local reconnection. In: Thompson JF, Soni BK, Weatherill NP, editors. Handbook of grid generation. Boca Raton: CRC Press; 1999. p. 18-1-18-31.

[28] Bocca P, Carpinteri A, Valente S. Mixed mode fracture of concrete. Int J Solids Struct 1991;27:1139–53.

[29] Zhu JZ, Zienkiewicz OC. Adaptive techniques in the finite element method. Commun Appl Numer Math 1988;4:197–204.