Computers & Structures Vol. 47, No. 3, pp. 441–450, 1993
Printed in Great Britain.

0045-7949/93 $6.00 + 0.00

A FAST ALGORITHM FOR GENERATING CONSTRAINED DELAUNAY TRIANGULATIONS

S. W. SLOAN
Department of Civil Engineering and Surveying, University of Newcastle, Shortland, NSW 2308, Australia

(*Received* 4 *March* 1992)

Abstract—A fast algorithm for generating constrained two-dimensional Delaunay triangulations is described. The scheme permits certain edges to be specified in the final triangulation, such as those that correspond to domain boundaries or natural interfaces, and is suitable for mesh generation and contour plotting applications. Detailed timing statistics indicate that its CPU time requirement is roughly proportional to the number of points in the data set. Subject to the conditions imposed by the edge constraints, the Delaunay scheme automatically avoids the formation of long thin triangles and thus gives high quality grids. A major advantage of the method is that it does not require extra points to be added to the data set in order to ensure that the specified edges are present.

INTRODUCTION

Triangulation schemes are used in a variety of scientific applications including contour plotting, volume estimation, and mesh generation for finite element analysis. Some of the most successful techniques are undoubtedly those that are based on the Delaunay triangulation.

To describe the construction of a Delaunay triangulation, and hence explain some of its properties, it is convenient to consider the corresponding Voronoi diagram (which is also known as the Dirichlet tessellation). The Voronoi diagram for seven points in the plane is shown by the dotted lines in Fig. 1. Each Voronoi polygon is constructed by drawing perpendicular bisectors through the lines that connect each point to its nearest neighbours. Every polygon is associated uniquely with a single point and all sites within a polygon are closer to this point than any other point in the data set. Because of this last property, Voronoi diagrams are often used in geometric algorithms which require nearest neighbour searching.

Once the Voronoi diagram for a set of points is known, the corresponding Delaunay triangulation is readily computed by connecting all pairs of points which share a polygon boundary (see Fig. 1). Generally speaking, each vertex of the Voronoi diagram is located at the point of contact of three adjacent polygons and, hence, defines the circumcentre for a Delaunay triangle. This last condition implies that the circumcircle for each triangle is 'empty', since it may not contain a vertex. Except in isolated instances, the Delaunay triangulation associated with an arbitrary set of points is unique. One important case of a nonunique triangulation occurs for a set of four points which lie exactly on the vertices of a square, as shown in Fig. 2. This arrangement is said to be degenerate since two valid Delaunay triangulations are possible. Although it leads to a loss of uniqueness, degeneracy is seldom a cause for concern since a triangulation may always be generated by making an arbitrary, but consistent, choice between two alternative patterns.

One major advantage of the Delaunay triangulation, as opposed to a triangulation constructed heuristically, is that it automatically avoids the creation of long thin triangles, with small included angles, wherever this is possible. Indeed, Lawson [1] has proved that the Delaunay triangulation is, by definition, locally equiangular. This means that for every convex quadrilateral formed by two adjacent triangles, the minimum of the six angles in the two triangles is greater than it would have been if the alternative diagonal had been drawn and the other pair of triangles chosen. Because of this property, Delaunay triangulations are a natural choice for mesh generation in finite element analysis and contour plotting applications.

A number of algorithms for generating the Delaunay triangulations have been proposed, including [1–6]. Operation counts for each of these schemes, expressed in terms of the number of points in the data set N, are shown in Table 1. Note that the average-case counts are for the standard example of a set of points distributed randomly over a unit square and give some indication of each algorithm's expected performance on practical problems. The worst-case counts, on the other hand, are often derived from contrived examples, which rarely occur in practice, and do not necessarily indicate the overall usefulness of an algorithm. Although the first scheme of Lee and Schachter [3] is asymptotically the most efficient, it is difficult to implement and does not appear to have been widely used. Efficient FORTRAN implementations of the Watson [4] and Cline and Renka [5]

Table 1. Operation counts for various Delaunay schemes

Algorithm	Average case	Worst case
Cline and Renka [5]	$O(N^{4/3})$	$O(N^2)$
Green and Sibson [2]	$O(N^{3/2})$	$O(N^2)$
Lawson [1]	$O(N^{4/3})$	$O(N^2)$
Lee and Schachter [3] (1)	$O(N \log_2 N)$	$O(N \log_2 N)$
Lee and Schachter [3] (2)	$O(N^{3/2})$	$O(N^2)$
Sloan [6]	$O(N^{5/4})$	$O(N^2)$
Watson [4]	$O(N^{3/2})$	$O(N^2)$

algorithms may be found, respectively, in Sloan and Houlsby [7] and Renka [8]. Both of these algorithms have been employed widely in finite element applications. A feature of Renka's program [8] is that it minimizes the storage needed to hold the triangulation by using a very compact data structure. This advantage, however, is gained at the expense of introducing additional complexity into the code with a subsequent loss in speed of execution. Another FORTRAN program, which combines some of the better features of the Lawson and Watson procedures, has been described by Sloan [6]. This paper pays special attention to efficiency, and the performance of the proposed scheme is compared directly with the above-mentioned implementations. For the test case of 10,000 points distributed randomly throughout a unit square, Sloan's procedure is just over four times faster than Sloan and Houlsby's code and slightly less than three times faster than Renka's code. When the number of points in the data set is 1000, this speed-up is reduced to a factor of around 1.6 for both cases. At the cost of some loss in modularity, the performance of Sloan's scheme may be further enhanced by coding the most frequently used subroutines 'in-line'. This removes the substantial overhead that is inherent in FORTRAN calls to short subroutines and improves the performance of the implementation by a factor of between

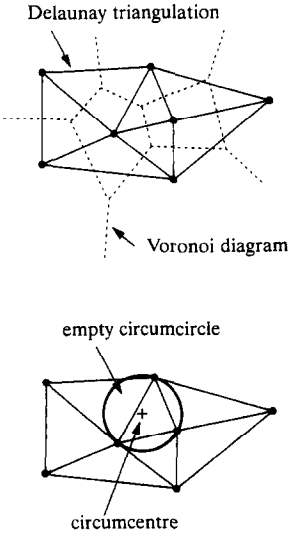


Fig. 1. The Delaunay triangulation.

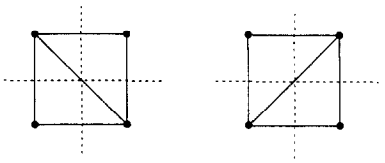


Fig. 2. A degenerate Delaunay triangulation.

two and three. Numerical experiments suggest that the run time for the algorithm is, for all practical purposes, directly proportional to N. This is substantially less than the expected growth rate of $O(N^{5/4})$ shown in Table 1, which is not reached until N is greater than several hundred thousand.

Many of the commonly cited Delaunay schemes have been developed to triangulate the convex hull of a set of points, and are not designed to incorporate constrained edges. This capability is highly desirable in finite element and terrain modelling applications, where the user must be able to specify natural boundaries and interfaces, but has proved to be difficult to implement in an elegant and efficient manner. In the FORTRAN program of Renka [8], a facility is provided for adjusting an existing triangulation so that specified edges are forced to be present. This important feature is, unfortunately, undocumented in the original paper of Cline and Renka [5] and no description of the algorithm, apart from comments in the source code, appears to be available. A different approach for including edge constraints has been proposed by De Floriani *et al.* [9], who first compute the Delaunay triangulation for the points on the domain boundary (at all times preserving its integrity) and then add the internal points by incrementally updating each existing triangulation. Whilst operation counts suggest that this strategy is efficient for simply connected regions, it is not suited to the more general problem of multiply connected domains. Another algorithm, due to Lee and Lin [10], tackles the problem of constrained edges by using a visibility graph during the assembly of the Delaunay triangulation. Although their scheme can incorporate any number of boundaries of arbitrary shape, it is complex to implement and typically requires $O(N^2)$ operations to construct the triangulation. Using a more sophisticated divide-and-conquer strategy, Chew [11] has proposed a constrained Delaunay scheme which is 'optimal' in the sense that it has a worst-case operation count of $O(N \log_2 N)$. This procedure can theoretically deal with multiply connected domains of arbitrary shape, but also appears to be rather complex and no performance or implementation details are provided.

In the finite element literature, a number of Delaunay schemes for generating meshes of arbitrary shape have recently been proposed. Schroeder and Shepard [12], for example, proceed by first constructing an initial unconstrained triangulation which is then adjusted to incorporate the required boundaries. Any holes in the boundary are 'stitched' together

by using additional points to update the initial triangulation. Weatherill [13] employs a similar strategy except that the points defining each boundary are, if necessary, supplemented with additional points prior to computing the Delaunay triangulation. This procedure requires an initial scan of the boundary points and connectivities to ensure that the resulting grid satisfies the specified boundaries. In a completely different approach, developed by Lo [14], the constrained Delaunay triangulation is assembled as an advancing front and additional points are not required. Lo advocates the use of a smoothing procedure to enhance the quality of the final triangulation and his test results suggest that, for problems of modest size, the overall CPU time of the algorithm is directly proportional to N.

This paper describes an efficient algorithm for computing constrained Delaunay triangulations. The scheme may be used to triangulate two-dimensional bodies of arbitrary shape and is particularly suited to mesh generation for contour plotting and finite element analysis. The algorithm first assembles a simple unconstrained triangulation and then updates this, if necessary, to force each of the constrained edges to be present. All of the edge constraints can be satisfied without inserting additional points into the triangulation. Detailed statistics indicate that the CPU time of the scheme is, for all practical purposes, directly proportional to the number of points in the data set.

ALGORITHM FOR CONSTRUCTING A DELAUNAY TRIANGULATION

In the first stage of our constrained Delaunay scheme, the algorithm of Sloan [6] is used to construct a simple unconstrained triangulation for the points. This initial grid does not necessarily include all of the desired edges and needs to be updated in the second phase. As discussed previously, Sloan's scheme is considerably faster than the Watson [4] and Cline and Renka [5] schemes and is also numerically robust. It has the further advantage in that it generates, as a by-product, an adjacency list for each of the triangles. This information, together with the list of vertices for each triangle, completely specifies the triangulation in a manner which is most convenient for finite element and contouring applications. The basic steps in the algorithm are as follows:

1. (Normalize coordinates of points.) Scale the coordinates of the points so that they all lie between 0 and 1. This scaling should be uniform so that the relative positions of the points are unchanged.

2. (Sort points into bins.) Cover the region to be triangulated by a rectangular grid so that each rectangle (or bin) contains roughly $N^{1/2}$ points. Label the bins so that consecutive bins are adjacent to one another, for example by using column-by-column or row-by-row ordering, and then allocate each point to its appropriate bin. Sort the list of points in ascending sequence of their bin numbers so that consecutive points are grouped together in the x–y plane.

3. (Establish the supertriangle.) Select three dummy points to form a supertriangle that completely encompasses all of the points to be triangulated. This supertriangle initially defines a Delaunay triangulation which is comprised of a single triangle. Its vertices are defined in terms of normalized coordinates and are usually located at a considerable distance from the window which encloses the set of points.

4. (Loop over each point.) For each point P in the list of sorted points, do steps 5–7.

5. (Insert new point in triangulation.) Find an existing triangle which encloses P. Delete this triangle and form three new triangles by connecting P to each of its vertices. The net gain in the total number of triangles after this stage is two. The searching algorithm of Lawson [1] may be used to locate the triangle containing P efficiently. Because of the bin sorting phase, only a few triangles need to be examined if the search is initiated in the triangle which has been formed most recently.

6. (Initialize stack.) Place all triangles which are adjacent to the edges opposite P on a last-in–first-out stack. There is a maximum of three such triangles.

7. (Restore Delaunay triangulation.) While the stack of triangles is not empty, execute Lawson's swapping scheme, as defined by steps 7.1–7.3.

   7.1. Remove a triangle which is opposite P from the top of the stack.

   7.2. If P is outside (or on) the circumcircle for this triangle, return to step 7.1. Else, the triangle containing P as a vertex and the unstacked triangle form a convex quadrilateral whose diagonal is drawn in the wrong direction. Swap this diagonal so that two old triangles are replaced by two new triangles and the structure of the Delaunay triangulation is locally restored.

   7.3. Place any triangles which are now opposite P on the stack.

In step 1, the coordinates for each of the points are normalized using the simple formulae

$$\hat{x} = (x - x_{\min})/d_{\max}, \qquad \hat{y} = (y - y_{\min})/d_{\max},$$

where $d_{\max} = \max\{x_{\max} - x_{\min},\ y_{\max} - y_{\min}\}$ is the maximum dimension of the bounding window. This scaling ensures that all of the coordinates are between 0 and 1 but does not modify the relative positions of the points in the x–y plane. The use of normalized coordinates, although not essential, reduces the effects of roundoff error and is also convenient from a computational point of view.

A key feature of the above scheme is that it is essentially recursive, with each point being introduced, one at a time, into an existing Delaunay triangulation. To update the triangulation at each

stage, we first search the grid, using the algorithm of Lawson [1], to find an existing triangle which encloses the new point. This enables the point to be inserted in the triangulation and the Delaunay condition is then restored by using a triangle-swapping algorithm, also due to Lawson [1]. For the searching phase to be fast, it is essential that a minimum number of triangles are inspected, especially for large problems. The purpose of step 2 is to sort the points which are close together into clusters, with roughly $N^{1/2}$ points in each cluster, so that the Lawson search is very efficient. By initiating the search in the triangle that has been formed most recently, the Lawson procedure typically examines only $O(N^{1/4})$ triangles before the search is terminated. One possible method of arranging the bins for sorting the points into clusters is shown in Fig. 3. In this scheme, the bin number for each point is given by

$$b = i \times n + j + 1, \quad \text{for } i \text{ even}$$

$$b = (i + 1)n - j, \quad \text{for } i \text{ odd},$$

where the row and column indices (i, j) are computed from

$$i = \text{int}(0.99 \times n \times \hat{y}/\hat{y}_{\max})$$

$$j = \text{int}(0.99 \times n \times \hat{x}/\hat{x}_{\max})$$

and n, the number of bins in the x- and y-directions, is equal to $N^{1/4}$ (to the nearest integer). The factor of 0.99 is required to ensure that points with the maximum coordinates do not fall outside the grid. Once they have all been assigned to their appropriate bins, the points may be grouped into the required clusters by sorting them in ascending sequence of their bin numbers. Because all of the bin numbers, which act as 'keys', are integers that lie in a known range, the pocket sort of Knuth [15] may be used to order the points in only $O(N)$ operations. This step was originally implemented using the quicksort algorithm [6], but this requires $O(N \log_2 N)$ operations and is slower than the pocket sort algorithm by a factor of around three. It is perhaps worth remarking that the bin sorting phase is an optional component of the triangulation scheme. The results reported in [6], however, suggest that this step is usually worthwhile as it adds only a slight overhead and typically reduces the execution time of the algorithm significantly. We will examine this question in more detail in a later section.

In step 3, it is convenient to number the vertices of the supertriangle as $N+1$, $N+2$ and $N+3$ and allocate them normalized coordinates of $(-100, -100)$, $(100, -100)$ and $(0, 100)$, respectively. The size and shape of the supertriangle may be chosen arbitrarily, the only restriction being that it must enclose all of the points in the data set. Note that if the vertices of the supertriangle are very close to the bounding window for the set of points, the outer boundary of the resulting unconstrained Delaunay triangulation may be not be strictly convex. This is not an important issue for most applications and may be resolved by simply relocating the vertices at a greater distance from the bounding window. The use of the supertriangle, which was first proposed by Watson [4], ensures that each point can be inserted inside an existing grid, and permits the Delaunay triangulation to be constructed in a simple and elegant manner.

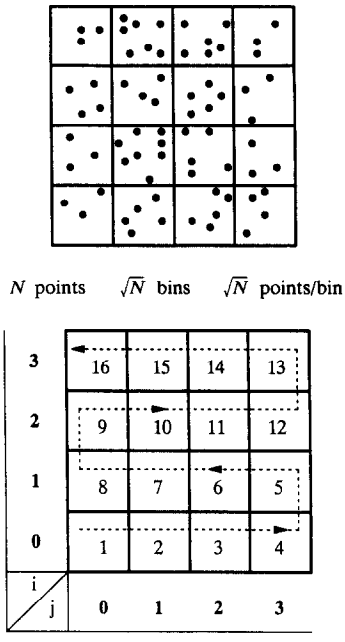


Fig. 3. Bin sorting procedure.

The data structure used to hold the triangulation is shown in Fig. 4. The vertices for each triangle are listed anticlockwise and stored in a single column of a two-dimensional array V. Similarly, the list of adjacent triangles are held in the two-dimensional

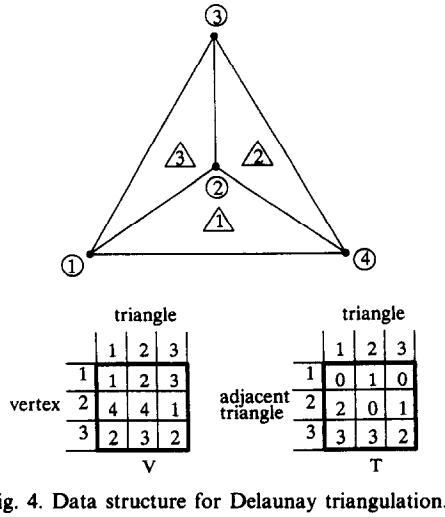


Fig. 4. Data structure for Delaunay triangulation.

array T, with a zero denoting that the side lies on a boundary.

In the first stage of step 5, it is necessary to locate a triangle which encloses the new point P. The search is initiated in the triangle which was formed last and uses Lawson's algorithm to march from one triangle to the next in the general direction of P. This ingenious strategy removes the need to search through the entire triangulation, and is shown in Fig. 5. Once the enclosing triangle is found, it is deleted and the new point is inserted into the grid by connecting P to each of its vertices. This process gives a net gain in the total number of triangles of two and is illustrated in Fig. 6. Since we start out with a single supertriangle, and each new point creates two new triangles, the final number of triangles is equal to $2N + 1$.

In steps 6 and 7, the triangulation is restored to a Delaunay triangulation using Lawson's swapping algorithm. After the insertion of P, all the triangles which are now opposite P are placed on a stack, as shown in Fig. 6. Each triangle is then removed from the stack, one at a time, and a check is made to see if P lies inside its circumcircle. If this is the case, then the two triangles which share the edge opposite P violate the Delaunay condition and form a convex quadrilateral with the diagonal drawn in the wrong direction. To satisfy the Delaunay constraint that each triangle has an empty circumcircle, the diagonal of the quadrilateral is simply swapped, and any triangles which are now opposite P are placed on the stack. This process is repeated until the stack is empty, which signals that the triangulation has been restored to a Delaunay triangulation, and is illustrated in Fig. 7. When implementing Lawson's swapping algorithm, it is essential that the circumcircle test is computed efficiently and accurately, as this step accounts for a significant proportion of the total CPU time and determines the validity of the triangulation. A fast and numerically robust method for performing this test has been given by Cline and Renka [5]. Consider the two adjacent triangles shown

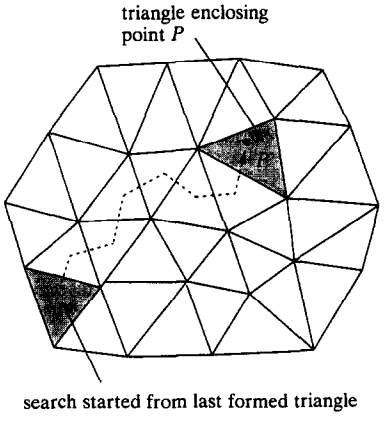


Fig. 5. Lawson's search.

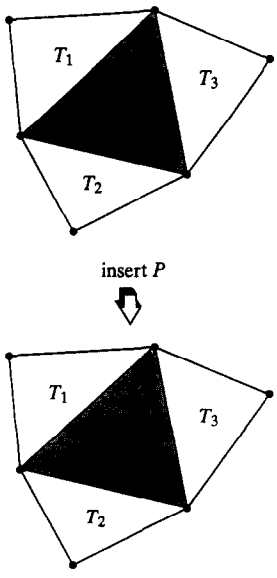


Fig. 6. Insertion of new point in triangulation.

in Fig. 8. The Delaunay constraint stipulates that the diagonal V_1–V_2 is replaced by the diagonal P–V_3 if P lies inside the circumcircle for the triangle V_1–V_2–V_3. From Fig. 8 we see that P lies on the circumcircle when $2\alpha + 2\beta = 2\pi$ and, thus, a swap must be performed when $\alpha + \beta > \pi$. Since $\alpha + \beta < 2\pi$, the swap condition also implies that

$$\sin(\alpha + \beta) = \cos\alpha \sin\beta + \sin\alpha \cos\beta < 0$$

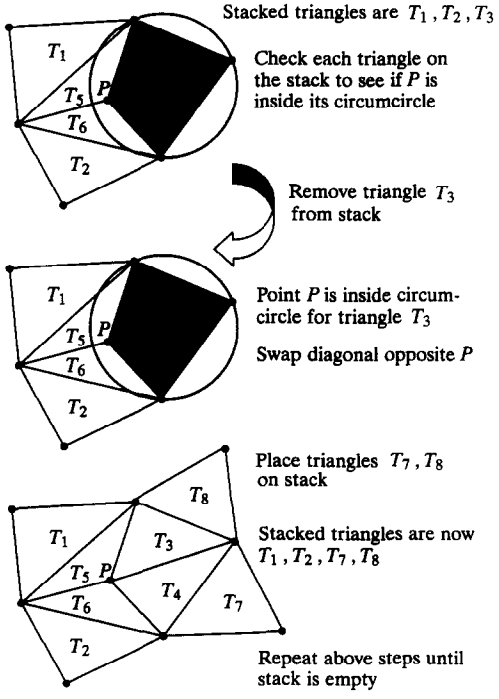
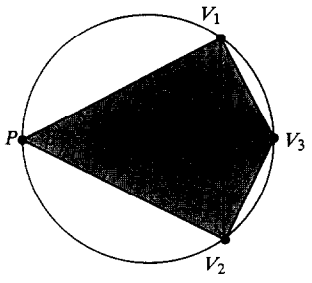


Fig. 7. Lawson's swapping algorithm.



$\alpha + \beta = \pi$

Fig. 8. Geometry for circumcircle test.

which may also be written as

$$\frac{(x_{13}x_{23} + y_{13}y_{23})(x_{2P}y_{1P} - x_{1P}y_{2P}) + (x_{13}y_{23} - x_{23}y_{13})(x_{2P}x_{1P} + y_{2P}y_{1P})}{[(x_{13}^2 + y_{13}^2)(x_{23}^2 + y_{23}^2)(x_{2P}^2 + y_{2P}^2)(x_{1P}^2 + y_{1P}^2)]^{1/2}} < 0,$$

where

$$x_{13} = x_1 - x_3, \qquad y_{13} = y_1 - y_3,$$

$$x_{23} = x_2 - x_3, \qquad y_{23} = y_2 - y_3,$$

$$x_{1P} = x_1 - x_P, \qquad y_{1P} = y_1 - y_P,$$

$$x_{2P} = x_2 - x_P, \qquad y_{2P} = y_2 - y_P.$$

Multiplying through by the denominator and rearranging, this test may be expressed in the more efficient form

$$(x_{13}x_{23} + y_{13}y_{23})(x_{2P}y_{1P} - x_{1P}y_{2P}) < (x_{23}y_{13} - x_{13}y_{23})(x_{2P}x_{1P} + y_{1P}y_{2P}) \quad (1)$$

which requires only ten multiplications, two additions and two subtractions. Although the circumcircle test of eqn (1) is very fast, the effects of round-off error may cause it to become inaccurate when $\sin(\alpha + \beta)$ approaches zero. This condition arises when:

(i) $\alpha + \beta$ is near π,
(ii) α and β are both near 0,
(iii) α and β are both near π,

and has been discussed at length in [5]. The first case indicates that P lies very close to, or precisely on, the circumcircle for the triangle V_1–V_2–V_3 and may be safely ignored. The second and third cases, on the other hand, occur when the vertices of the quadrilateral V_1–P–V_2–V_3 are very nearly collinear and must be dealt with to preserve the correctness of the triangulation. In particular, it is necessary to always perform a swap when α and β are both near π. In light of these considerations, Cline and Renka [5] suggested that the circumcircle test of eqn (1) should be modified as follows:

(i) Set $\cos a = x_{13}x_{23} + y_{13}y_{23}$ and $\cos b = x_{2P}x_{1P} + y_{2P}y_{1P}$.
(ii) If $\cos a \geq 0$ and $\cos b \geq 0$ then set the swap test to 'false' and exit.
(iii) If $\cos a < 0$ and $\cos b < 0$ then set the swap test to 'true' and exit.
(iv) Set $\sin ab = (x_{13}y_{23} - x_{23}y_{13})\cos b + (x_{2P}y_{1P} - x_{1P}y_{2P})\cos a$.
(v) If $\sin ab < 0$ then set the swap test to 'true' and exit.
(vi) Set the swap test to 'false'.

Although this revised procedure introduces a number of additional comparisons, it has the great advantage of being numerically stable.

The expected run time performance of the triangulation algorithm may be determined by analysing the complexity of each of its steps. As discussed previously, the bin sort in step 2 may be implemented so that it requires $O(N)$ operations. Even for large problems, only a small fraction of the total CPU time is spent in this phase and the bulk of the computational work is done by the searching and swapping procedure of steps 5 and 7. To estimate the expected behaviour of step 5, it is reasonable to assume that the points are distributed in the x–y plane so that each bin contains roughly $O(N^{1/2})$ points. For this case, $O(N^{1/4})$ triangles need to be searched to find the triangle which encloses the newly introduced point. Since this step is repeated N times, it gives rise to an overall operation count of $O(N^{5/4})$. Somewhat surprisingly, the swapping scheme in step 7 typically requires only a constant number of operations. Indeed, this step has a total operation count of $O(N)$, since numerical experiments suggest that the introduction of each new point requires an average of about three swaps to restore the Delaunay structure. Interestingly, the observed run times for Sloan's scheme usually grow at a rate which is substantially less than the theoretical prediction of $O(N^{5/4})$. This apparent discrepancy is due to the fact that the time required for one iteration of the search procedure is much less than the time required for one iteration of the swapping procedure. Even for large values of N, the average number of searches is small and most of the time is spent in the swapping phase. Unless N is indeed huge, the average run time of the algorithm is very nearly $O(N)$.

ALGORITHM FOR CONSTRUCTING A CONSTRAINED DELAUNAY TRIANGULATION

The Delaunay triangulation discussed in the previous section is unconstrained and does not necessarily include all of the prescribed edge constraints. In finite element and contouring applications, these edge constraints typically correspond to domain boundaries or natural discontinuities

(such as an interface between two different types of material). The algorithm for modifying the existing Delaunay triangulation, so that certain edges are forced to be present, is as follows:

1. (Loop over each constrained edge.) Let each constrained edge be defined by the vertices V_i and V_j. For each of these edges, do steps 2–4.
2. (Find intersecting edges.) If the constrained edge V_i–V_j is already present in the triangulation, then go to step 1. Else, search the triangulation and store all of the edges that cross V_i–V_j.
3. (Remove intersecting edges.) While some edges still cross the constrained edge V_i–V_j, do steps 3.1 and 3.2.
   - 3.1. Remove an edge from the list of edges that intersect V_i–V_j. Let this edge be defined by the vertices V_k and V_l.
   - 3.2. If the two triangles that share the edge V_k–V_l do not form a quadrilateral which is strictly convex, then place V_k–V_l back on the list of intersecting edges and go step 3.1. Else, swap the diagonal of this strictly convex quadrilateral so that two new triangles are substituted for two old triangles. Let the new diagonal be defined by the vertices V_m and V_n. If V_m–V_n still intersects the constrained edge V_i–V_j, then place it on the list of intersecting edges. If V_m–V_n does not intersect V_i–V_j, then place V_m–V_n on a list of newly created edges.
4. (Restore Delaunay triangulation.) Repeat steps 4.1–4.3 until no further swaps take place.
   - 4.1. Loop over each edge in the list of newly created edges.
   - 4.2. Let the newly created edge be defined by the vertices V_k and V_l. If the edge V_k–V_l is equal to the constrained edge V_i–V_j, then skip to step 4.1.
   - 4.3. If the two triangles that share the edge V_k–V_l do not satisfy the Delaunay criterion, so that a vertex of one of the triangles is inside the circumcircle of the other triangle, then these triangles form a quadrilateral with the diagonal drawn in the wrong direction. In this case, the edge V_k–V_l is swapped with the other diagonal (say) V_m–V_n, thus substituting two new triangles for two new triangles, and V_k–V_l is replaced by V_m–V_n in the list of newly created edges.
5. (Remove superfluous triangles.) Remove all triangles that contain a supertriangle vertex or lie outside the domain boundary.

This algorithm is similar to the one used in the code in [5], although our data structure and method of implementation are somewhat different. The scheme is essentially comprised of two distinct phases. In the first phase, each constrained edge is forced to be present in the triangulation by using a triangle swapping procedure to remove all of the intersecting edges. The swapping algorithm is designed to test all possible arrangements and will always find a triangulation which includes the constrained edge. In the second phase, each new edge that is formed by the triangle swapping process, except the constrained edge, is 'optimized' so that it satisfies the Delaunay condition. Subject to the restrictions imposed by the edge constraints, which may force some triangles to have non-empty circumcircles, the final grid is thus a Delaunay triangulation.

A rigorous complexity analysis of the scheme is difficult because the number of operations required depends inevitably on the problem at hand. We may, however, make some preliminary remarks which are based on empirical observations.

In step 2, it is necessary to examine an average of about three edges to determine whether the prescribed edge is present in the triangulation. To make this step efficient, it is convenient to hold a list of triangle numbers for each vertex, so that we can readily locate a triangle to which a given vertex belongs. This vector must be of length $N + 3$, and provides a starting triangle to begin the search for the edges which cross the constrained edge V_i–V_j. To find all the intersections, we circle the vertex V_i until the first crossing is detected and then march from one triangle to the next in the general direction of the vertex V_j. This procedure, shown in Fig. 9, is efficient since each intersecting edge needs to be inspected only once to form the list of crossings.

The iterative algorithm of step 3, which rearranges the grid to incorporate the edge V_i–V_j, hinges on the fact that all possible triangulations for a set of points can be found by systematically swapping the diagonal in each convex quadrilateral formed by a pair of triangles. For the case of a constrained edge which is crossed by n edges, it is clear that at least n swaps must be performed. Empirical evidence suggests that one iteration of this step typically reduces the number of crossings by a factor of two and, hence, it is usual to observe that roughly $O(\log_2 n)$ iterations in total are required. The algorithm for removing all of the edges which intersect a constrained edge is illustrated in Fig. 10.

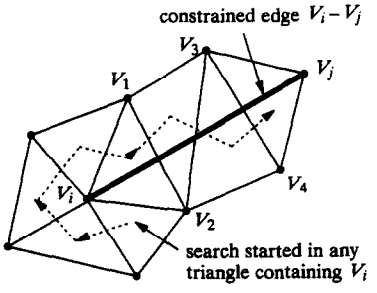


Fig. 9. Detecting constrained edge intersections.

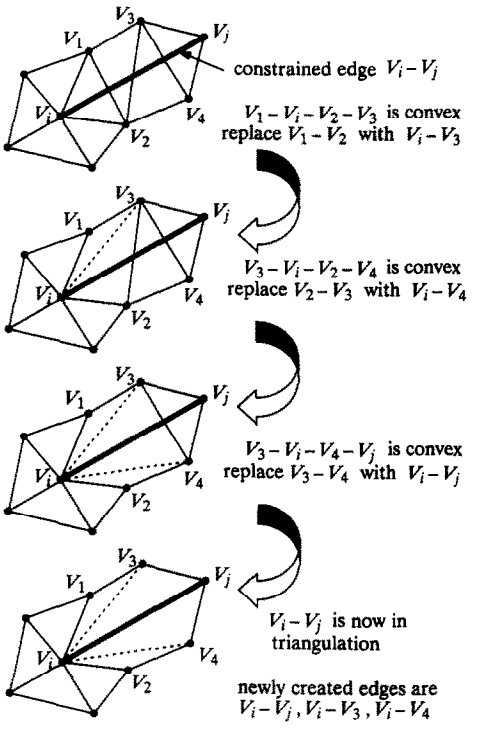


Fig. 10. Removing constrained edge intersections.

The optimization iterations of step 4, shown in Fig. 11, serve to restore the Delaunay condition for each new edge created in step 3, except for the constrained edge, and ensure that the overall triangulation is of good quality. For most practical problems this phase requires only two or three sweeps through the list of newly created edges and, thus, typically gives rise to an operation count of $O(n)$.

The task of clipping unwanted triangles from the grid, which is the final stage in the triangulation scheme, may be implemented so that only $O(N)$ operations are required. If the constrained edges define domain boundaries, we first trace each boundary and mark every triangle that needs to be deleted. These marked triangles are then removed in a single pass and it is unnecessary to inspect every triangle in the grid.

APPLICATIONS

The constrained triangulation scheme described in the previous sections has been implemented in standard FORTRAN 77. For an arbitrary triangulation with N points and N_c constrained edges, the code requires $17N + 2N_c + 18$ words of memory, including the storage for holding the coordinates, and has been used successfully on data sets with several hundred thousand points. After computing the triangulation, each triangle that does not share a constrained edge is checked to ensure that it satisfies the Delaunay condition by having an empty circumcircle. In addition, every triangle must have a positive area and the triangulation as a whole must obey Euler's formula for planar graphs. For a multiply connected domain with H 'holes', the latter condition may be written as $N + N_t + H - N_e = 1$, where N_t and N_e denote, respectively, the total number of triangles and edges in the grid. Numerous other checks, such as a test for coincident points, are also included to verify the integrity of triangulation.

To give some indication of the efficiency of the algorithm, we consider a set of N points distributed randomly over a unit square. For various values of N, ranging from 100 to 20,000, we compute both a simple Delaunay triangulation and a constrained Delaunay triangulation. The latter cases are obtained from the former cases merely by imposing $N/10$ edge constraints, which are selected at random, on the data set. Since they often span across the domain and give rise to a large number of intersections, these constrained edges pose a severe test for the scheme. In most practical applications, particularly in finite element analysis, it is usual for each constrained edge to be intersected by only a few edges.

Figures 12 and 13 illustrate, respectively, the unconstrained and constrained Delaunay triangulations for the case of 100 points. In Fig. 13, the constrained edges are indicated by bold lines. The results for the various test cases are shown in Table 2. These timing statistics were obtained using the internal clock of the machine, and include the time required to check both the data and the triangulation.

When the bin sort is used, the growth in the CPU time for the unconstrained triangulation scheme is very nearly linear with N. Indeed, averaging the results for the values of N considered, the observed

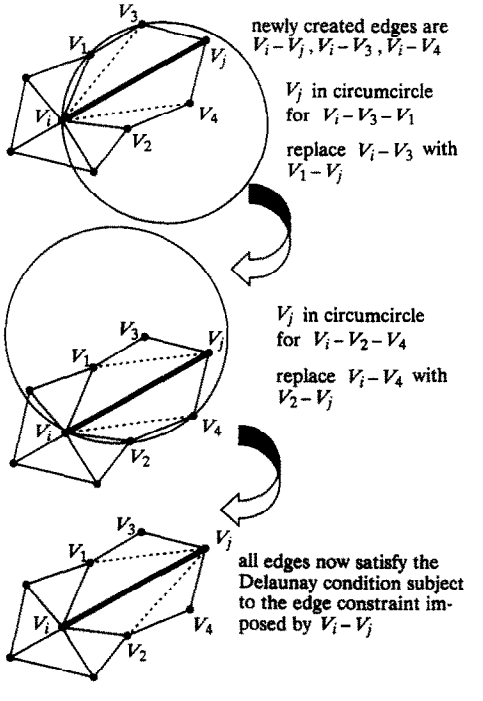


Fig. 11. Restoring constrained Delaunay condition.

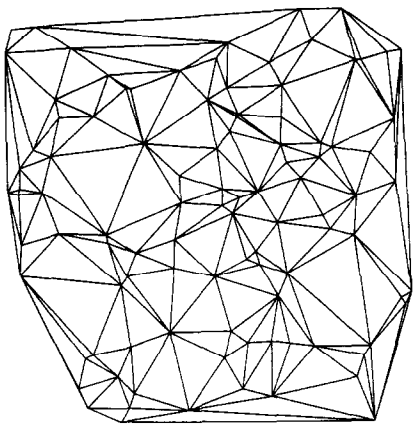


Fig. 12. Delaunay triangulation for 100 points.

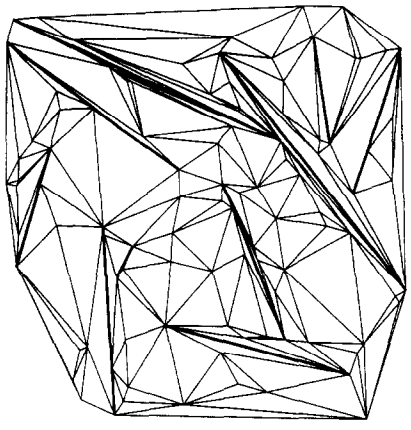


Fig. 13. Delaunay triangulation for 100 points with 10 random edge constraints.

run times are approximately $O(N^{1.06})$. For the constrained cases, with $N/10$ arbitrary edge constraints, the observed growth rate is slightly greater than linear and has an average value of $O(N^{1.12})$.

Without the bin sort, the observed run times for the unconstrained and constrained schemes may be respectively approximated as $O(N^{1.29})$ and $O(N^{1.27})$. When the points are distributed over the domain in a reasonably uniform manner, the importance of the bin sorting procedure increases with increasing N. For an N value of 1000, the bin sort reduces the run time of the algorithm by roughly 20%. This reduction increases to around 50% once N reaches 20,000.

For our model problem, the imposition of the constrained edges typically increases the CPU time, relative to that of the corresponding unconstrained case, by a factor of around 2. The time required to incorporate each constrained edge is, as discussed previously, dependent on how many times it is crossed by other edges in the initial triangulation. Since the edge constraints have been generated randomly and often intersect a lot of edges in the unconstrained triangulation, these test cases are especially severe and unlikely to occur in practice. In the majority of applications encountered by the author, the overhead associated with incorporating realistic edge constraints is typically less than 10%.

As a final example, the constrained Delaunay scheme was used to generate the illustrative finite

Table 2. Timing statistics for triangulation of points distributed randomly over a unit square

	Unconstrained Delaunay triangulation		Constrained Delaunay triangulation	
N	CPU time	a	CPU time	a
100	0.10		0.11	
	0.91		**0.12**	
		1.00		1.11
		1.16		**1.17**
1000	0.99		1.42	
	1.29		**1.77**	
		1.10		1.17
		1.30		**1.30**
5000	5.82		9.38	
	10.41		**14.29**	
		1.08		1.16
		1.35		**1.33**
10,000	12.28		20.98	
	26.55		**35.94**	
		1.07		1.04
		1.36		**1.27**
20,000	25.74		43.10	
	68.10		**86.69**	

Notes:

1. All times in CPU seconds for Apollo DN3500 (25 MHz Motorola 68030 with Weitek floating point accelerator).
2. Values for the exponent a obtained by assuming CPU times are $O(N^a)$.
3. Constrained Delaunay triangulations have $N/10$ edges prescribed randomly.
4. **Bold face** entries are for runs without the bin sort.

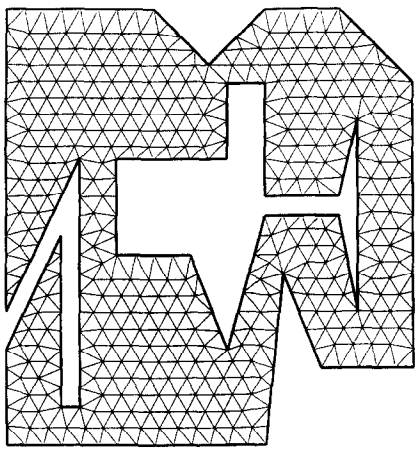


Fig. 14. Illustrative finite element mesh with 509 points and 818 triangles.

element mesh shown in Fig. 14. The grid has 509 points, 818 triangles and required 0.55 seconds of CPU time to generate on an Apollo DN3500.

REFERENCES

1. C. L. Lawson, Software for C^1 surface interpolation. In *Mathematical Software III* (Edited by J. R. Rice), pp. 161–194. Academic Press, New York (1977).
2. P. J. Green and R. Sibson, Computing Dirichlet tessellations in the plane. *Computer Jnl* **21,** 168–173 (1978).
3. D. T. Lee and B. J. Schachter, Two algorithms for constructing a Delaunay triangulation. *Int. J. Computer Inform. Sci.* **9,** 219–242 (1980).
4. D. F. Watson, Computing the n-dimensional Delaunay triangulation with application to Voronoi polytopes. *Computer Jnl* **24,** 167–172 (1981).
5. A. K. Cline and R. Renka, A storage efficient method for construction of a Thiessen triangulation. *Rocky Mountain J. Math.* **14,** 119–139 (1984).
6. S. W. Sloan, A fast algorithm for constructing Delaunay triangulations in the plane. *Advances in Engng Software* **9,** 34–55 (1987).
7. S. W. Sloan and G. T. Houlsby, An implementation of Watson's algorithm for computing two-dimensional Delaunay triangulations. *Advances in Engng Software* **6,** 192–197 (1984).
8. R. L. Renka, Algorithm 624: Triangulation and interpolation at arbitrarily distributed points in the plane. *ACM Trans. Math. Software* **10,** 440–442 (1984).
9. L. De Floriani, B. Falciendo and C. Pienovi, Delaunay-based representation of surfaces defined over arbitrarily shaped domains. *Computer Vision, Graphics and Image Processing* **32,** 127–140 (1985).
10. D. T. Lee and A. K. Lin, Generalized Delaunay triangulations for planar graphs. *Discrete and Computational Geometry* **1,** 201–217 (1986).
11. L. P. Chew, Constrained Delaunay triangulations. *Algorithmica* **4,** 97–108 (1989).
12. W. J. Schroeder and M. S. Shepard, Geometry-based fully automatic mesh generation and the Delaunay triangulation. *Int. J. Numer. Meth. Engng* **26,** 2503–2515 (1988).
13. N. P. Weatherill, The integrity of geometrical boundaries in the two-dimensional Delaunay triangulation. *Commun. Appl. Numer. Meth.* **6,** 101–109 (1990).
14. S. H. Lo, Delaunay triangulation of non-convex planar domains. *Int. J. Numer. Meth. Engng* **28,** 2695–2707 (1989).
15. D. E. Knuth, *Sorting and Searching: The Art of Computer Programming III*, pp. 170–178. Addison-Wesley, MA (1973).