A fast algorithm for constructing Delaunay triangulations in the plane

S. W. SLOAN

Department of Civil Engineering and Surveying, The University of Newcastle, NSW 2308, Australia

This paper describes an algorithm for computing Delaunay triangulations of arbitrary collections of points in the plane. A FORTRAN 77 implementation of the scheme is given. For \( N \) points distributed randomly within a square domain, the expected run time for the algorithm is approximately \( O(N^{2/4}) \). Empirical tests, for \( N \) up to 10,000, indicate that the actual run time is substantially less than this prediction and is generally better than \( O(N^{1.1}) \). Excluding the memory required to store the co-ordinates, the algorithm requires slightly greater than \( 14N \) words of integer memory to complete a typical triangulation. The efficiency of the proposed algorithm is verified by comparing its performance with other Delaunay triangulation procedures. Uses of the algorithm include the generation of finite element meshes and the construction of contour plots.

Key Words: Delaunay, triangulation, algorithm.

INTRODUCTION

The problem of triangulating arbitrary collections of points in the plane occurs frequently in engineering and examples include mesh generation for finite element analysis and the construction of contour plots.

The theory of Delaunay triangulations has been described previously but will be discussed briefly for completeness. To describe the construction of a Delaunay triangulation it is convenient to consider the corresponding Dirichlet tessellation. The Dirichlet tessellation for five points in the plane is shown in Fig. 1 and is denoted by the heavy lines. This tessellation divides the plane into a collection of polygonal regions whose boundaries are the perpendicular bisectors of the lines joining the neighbouring data points. Each polygon is associated with a single data point. Any location within a given polygon is closer to the polygon data point than any other data point. The Delaunay triangulation that corresponds to the Dirichlet tessellation is constructed by connecting all data points that share a polygon boundary. The Delaunay triangulation for the five points in Fig. 1 is indicated by the faint lines.

In general, the vertices of the Dirichlet tessellation occur where three adjacent polygons meet. The three data points associated with each of these polygons form a Delaunay triangle. By definition, each vertex of a Dirichlet tessellation is equidistant from each of the three data points forming the Delaunay triangle. Thus, each vertex of the Dirichlet tessellation is uniquely associated with a Delaunay triangle and is located at its circumcentre. When the Delaunay triangulation is complete, this means that no data point may lie inside the circumcircle of any triangle.

Generally speaking, the Delaunay triangulation associated with an arbitrary set of points in the plane is unique. In some instances, however, the triangulation may not be unique and is said to be degenerate. A very simple example which illustrates a degenerate triangulation is shown in Fig. 2, where four data points are located at the vertices of a square. The single vertex of the Dirichlet tessellation is located at the centroid of the square where four polygons meet. Two different Delaunay triangulations are possible with this configuration and both are equally valid. In practical algorithms, the problem of degeneracy is easily dealt with by making an arbitrary choice between alternative triangulations and does not pose any serious difficulties.

Figure 1. The Dirichlet tessellation and del-triangulation
One of the advantages of Delaunay triangulations, as opposed to triangulations constructed heuristically, is that they automatically avoid forming triangles with small included angles whenever this is possible. Indeed Lawson and Sibson have shown that Delaunay triangulations are, 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 particularly suited to grid generation for finite element analysis and contouring algorithms.

A number of algorithms for constructing planar Delaunay triangulations have been proposed. For a collection of \( N \) points, the average and worst case run times for the various algorithms are shown in Table 1. In engineering applications the average performance of a triangulation algorithm is generally more important than its worst case performance, since the latter tends to occur rarely in practice. Average run times for triangulation algorithms are usually deduced by considering collections of points located randomly within square or circular domains.

For large sets of points, the results in Table 1 indicate that the first algorithm of Lee and Schachter (these authors propose two algorithms) is the most efficient. This procedure, however, is complicated and difficult to implement. FORTRAN code for the Cline and Renka algorithm has been made publicly available by Renka and a simple FORTRAN 77 implementation of the Watson scheme is described in Sloan and Houlsby. Watson's algorithm, which is quite efficient for triangulating up to about 2000 points, has the advantage of being particularly simple. This paper describes a simple scheme which may be used to compute Delaunay triangulations for both small and large sets of points. Analysis of the algorithm indicates that its run time is \( O(N^{5/4}) \) for points that are distributed randomly within a square domain. Empirical comparisons with other procedures suggest that it is efficient.

**OUTLINE OF ALGORITHM**

The algorithm combines features of both the Watson and Lawson procedures. The Delaunay triangulation is assembled by introducing each point, one at a time, into an existing Delaunay triangulation which is then updated.

Following the idea of Watson, the process is started by selecting three points to form a 'supertriangle' which completely encompasses all of the data points to be triangulated. Initially the Delaunay triangulation is thus comprised of a single triangle defined by the supertriangle vertices. When a new point \( P \) is introduced into the triangulation, we first find an existing triangle which encloses \( P \) and form three new triangles by connecting \( P \) to each of its vertices. Note that during this step the original enclosing triangle is deleted and the net gain in the total number of triangles is two. After the new point \( P \) has been inserted, the existing triangulation is updated to a Delaunay triangulation using the swapping algorithm of Lawson. In this procedure all the triangles which are adjacent to the edges opposite \( P \) are placed on a last-in, first-out stack (ie. a maximum of three triangles are placed on the stack initially).

Each triangle is then unstacked, one at a time, and a check is made to determine if \( P \) lies inside its circumcircle. If this is the case then the triangle containing \( P \) as a vertex and the adjacent triangle form a convex quadrilateral with the diagonal drawn in the wrong direction, and it must be replaced by the alternative diagonal to preserve the structure of the Delaunay triangulation. The swapping procedure replaces two old triangles with two new triangles with no net gain in the total number of triangles. Once the swap is completed, any triangles which are now opposite \( P \) are added to the stack (there are a maximum of two). The next triangle is then unstacked and the whole process is repeated until the stack is empty and this results in a new Delaunay triangulation containing the point \( P \). An illustration of the swapping procedure is shown in Fig. 3. Note that if \( P \) lies outside (or on) the circumcircle for a stacked triangle, then no action is taken and we
Figure 3. Lawson's swapping algorithm

simply skip to the next triangle on the stack. It has been shown by Lawson that this iterative algorithm must result in a Delaunay triangulation and will always terminate after a finite number of swaps. Typically only a few levels of swaps are necessary for each edge which is initially opposite P and the process is thus efficient.

After all the points have been added to the triangulation, the final Delaunay triangulation is obtained by removing all of the triangles that contain one or more of the supertriangle vertices. Any vertex which appears in these deleted triangles, but is not a supertriangle vertex, must lie on the boundary of the triangulation. Since the insertion of each new point into the triangulation creates two new triangles the final number of triangles, including those formed with the vertices of the supertriangle, is 2N+1.

IMPLEMENTATION OF ALGORITHM

An implementation of the algorithm for computing Delaunay triangulations is given in Appendix 1. To the best of the author's knowledge the code strictly obeys the syntax of FORTRAN 77 and thus should be portable. The program uses single precision arithmetic which is considered to be satisfactory for computation on 32 bit machines. To convert the implementation to double arithmetic, all REAL declarations should be replaced by DOUBLE PRECISION declarations and all real constants in PARAMETER statements should be replaced by double precision constants. The program is comprised of five subroutines (DELTRI, BSORT, QSORTI, DELAUN, PUSH) and four short function subprograms (TRILOC, POP, EDG, SWAP). Each of these will be discussed in turn to illustrate the detail of the overall algorithm.

Subroutine DELTRI

This is the only subroutine that needs to be called by the user to construct the Delaunay triangulation and controls the overall flow of the program. When calling DELTRI, NUMPTS is the total number of points in the data set and N is the number of points to be triangulated. The set of points to be triangulated is stored in the integer vector LIST prior to calling the subroutine. LIST is of length N where N < NUMPTS. This allows the user to triangulate any subset of the total number of points and is particularly useful in practical applications. The co-ordinates of the points in the data set are stored in the real vectors X and Y. Each of these is of length NUMPTS + 3. The integer vector BIN is of length NUMPTS, and is required as auxiliary storage in subroutines BSORT and DELAUN. Throughout the program the structure of the Delaunay triangulation is stored in the integer arrays V and E. Both of these arrays are two-dimensional, with V containing the vertices for each triangle and E containing the adjacent triangles. The conventions for this data structure are shown for a simple example in Fig. 4. The dimensions of these arrays are V(3, 2*N + 1) and E(3, 2*N + 1).

At the beginning of subroutine DELTRI, the co-ordinates of the points to be triangulated are normalised to the values (x, y) according to

\[
\begin{align*}
\hat{x}_p &= (x_p - XMIN) / DMAX \\
\hat{y}_p &= (y_p - YMIN) / DMAX
\end{align*}
\]

where

\[
\begin{align*}
DMAX &= \text{MAX}(XMAX - XMIN, YMAX - YMIN) \\
XMIN &= \text{MIN}(x_p) \\
XMAX &= \text{MAX}(x_p) \\
YMIN &= \text{MIN}(y_p) \\
YMAX &= \text{MAX}(y_p)
\end{align*}
\]

This ensures that the values of \(\hat{x}\) and \(\hat{y}\) lie between 0 and 1 and proves convenient in the triangulation process.

After the triangulation has been computed, by calling the subroutines BSORT and DELAUN, subroutine DELTRI resets the co-ordinates of the points to their original values. Upon exiting from DELTRI, the Delaunay triangles are numbered from 1 to NUMTRI. Their vertex and adjacent triangle lists are stored in \(V(I, J)\) and \(E(I, J)\) where \(I = 1, 3\) and \(J = 1, \text{NUMTRI}\). Their vertex and adjacent triangle lists are stored in \(V(I, J)\) and \(E(I, J)\) where \(I = 1, 3\) and \(J = 1, \text{NUMTRI}\).

Subroutine BSORT

As described previously, the Delaunay triangulation is constructed by inserting each point, one at a time, into an existing triangulation. Before updating the triangulation, we first need to find an existing triangle which encloses the point to be inserted. The searching procedure used in the current algorithm is due to Lawson and is implemented in the function subprogram TRILOC (to be described later). This process is initiated at the triangle most recently

\[
\begin{align*}
\text{triangle} & \quad 1 \quad 2 \quad 3 \\
\text{vertices} & \quad 1 \quad 2 \quad 3 \\
\text{adjacent triangles} & \quad 0 \quad 1 \quad 0 \\
\text{0 denotes no adjacent triangle}
\end{align*}
\]

Figure 4. Data structure for triangulation algorithm
created and marches from one triangle to the next in the direction of the point to be inserted. The searching algorithm may be made efficient by presenting the points so that the distance between each point and its predecessor is small, thus reducing the number of triangles that need to be checked.

Following Lee and Schachter, we ensure that consecutive points are in close proximity by using a bin sort. In this procedure, which is implemented by subroutine BSORT, the region to be triangulated is covered by a grid of rectangles called bins. Each point is placed in a bin according to its co-ordinates and these bins are accessed as shown in Fig. 5. Assuming that the points are distributed uniformly in the \( x \)-\( y \) plane, we expect the number of points in each bin to be approximately equal. Empirical testing has indicated that it is sufficient to partition the domain to be triangulated into approximately \( N^{1/2} \) bins. Thus the number of bins in the \( x \)- and \( y \)-directions, \( NDIV \), is chosen as \( N^{1/4} \) (to the nearest integer). Since the normalised co-ordinates for the points lie in the range from 0 to 1, the number of bins in the \( x \)- and \( y \)-dimensions, \( NDIV \), is chosen arbitrarily. It is sufficient merely that it contains all of the points to be triangulated. It is worth noting, however, that if the corners of the supertriangle are very close to the window enclosing the points, the boundary of the final triangulation may be locally concave. Strictly speaking, such a triangulation does not correspond exactly to a Delaunay triangulation, since some long thin triangles along the boundary have been omitted. In most practical applications this is not a disadvantage, but may be avoided by locating the vertices of the supertriangle further away from the enclosing window.

After the supertriangle has been defined, subroutine DELAUN, which is the heart of the triangulation algorithm, inserts each of the points into the triangulation one at a time. To introduce a new point \( P \), a search is made to find an existing triangle \( T \) which encloses \( P \). Triangle \( T \) is then deleted and three new triangles are created by connecting \( P \) to each of its vertices (Fig. 6). Note that each of these triangles is created such that \( P \) is the first vertex in the vertex array. The net gain in the total number of triangles is two, and the additional triangles are numbered \( NUMTRI + 1 \) and \( NUMTRI + 2 \) where \( NUMTRI \) is the total number of triangles prior to the insertion of \( P \). Next, each triangle containing \( P \) as a vertex is placed on a last-in, first-out stack (provided that the edge opposite \( P \) is adjacent to some other triangle). With reference to Fig. 6, the triangles \( T \), \( NUMTRI + 1 \) and \( NUMTRI + 2 \) are placed on the stack and the triangles opposite \( P \) are respectively \( A \), \( B \) and \( C \). Note that it is unnecessary to maintain a separate stack of adjacent triangles which are opposite to \( P \), since these can be extracted from the adjacency arrays for the stacked triangles. For example, with reference to Fig. 6, the adjacent triangle which is opposite to \( P \) in triangle \( T \) is given by \( A = E(2, T) \). In general for element \( I \) in the stack, the opposite adjacent triangle is given by \( E(2, I) \). This completes the initial insertion phase in subroutine DELAUN, and we are now ready to update the triangulation to a Delaunay triangulation using the swapping algorithm of Lawson.
In Lawson's procedure, we remove each triangle from the stack one at a time. The notation used in subroutine DELAUN is shown in Fig. 7. Triangle L is the triangle removed from the stack. Triangle R is the triangle opposite point P which is adjacent to L. Triangles L and R share the edge V1 - V2 and form a quadrilateral with vertices P - V2 - V3 - V1 (triangle L is to the left of V2 - V1 and triangle R is to the right of V2 - V1). If the point P is inside the circumcircle of triangle R, then the diagonal V1 - V2 needs to be replaced with the diagonal P - V3 to preserve the structure of the Delaunay triangulation. As pointed out by Lawson, this circumcircle check maximises the minimum angle occurring in any pair of adjacent triangles forming a convex quadrilateral. If a swap is necessary, as shown in Fig. 7, the vertex and adjacent triangle lists are updated (again with P as the first vertex in the vertex array), as are the adjacency arrays for triangles L and R. Provided that there is a triangle opposite P which is adjacent to L, triangle L is placed on the stack. Similarly for triangle R. The next triangle is then removed from the stack and the whole process is repeated until the stack is empty. This signifies that the insertion of P is complete and the new triangulation is a Delaunay triangulation.

In subroutine DELAUN, the stacked triangles are stored in the vector STACK which has a length of NUMPTS. This dimension was found to be sufficient for triangulating 10,000 points distributed randomly within a square domain, but may be increased if the need arises.

After all of the points have been triangulated, subroutine DELAUN deletes any triangle which contains one or more supertriangle vertices. During this phase the vertex and adjacency arrays are updated to fill any blanks created and the Delaunay triangles are numbered from 1 to NUMTRI. The vertex and adjacent triangle lists are stored in V(I,J) and E(I,J) respectively, where I = 1, 3 and J = 1, NUMTRI.

Subroutine PUSH

This subroutine places an item on a last-in, first out stack and is easily understood. The maximum permissible size of the stack is MAXSTK, and this is defined in subroutine DELAUN. Subroutine PUSH includes a check to determine if there is sufficient space in STACK to complete the triangulation. If this is not the case, then a diagnostic message is printed and the execution of the program is halted.

Figure 6. Initial insertion of new point into triangulation

Figure 7. Implementation of Lawson's swapping algorithm
**Function POP**

This function removes an item from the top of a last-in, first-out stack and is complementary to subroutine PUSH. It includes a check to determine if the stack is already empty before attempting to remove an item. If this is so, a diagnostic message is printed and the execution of the program is halted.

**Function EDG**

This function finds the number of the edge (i.e. the row number in the triangle adjacency array) in element I which is adjacent to element J. Calls to this function should always result in a positive match. If elements I and J are found to be non-adjacent, function EDG prints a diagnostic message and terminates the execution of the program.

**Function TRILOC**

This function accepts the x-y co-ordinates of a point and finds an existing triangle which encloses it. The search is started at the triangle which has been most recently created, and checks if the point is to the right of any of its edges. Since the triangle vertices are always listed in an anticlockwise sequence, a point can only be enclosed by a triangle if it is to the left of each of its edges. (A point which lies on one of the edges of a triangle is also said to be enclosed by the triangle.) If the point lies to the right of any edge of the triangle, then the search shifts to the triangle which is adjacent to this edge and the process is repeated. In this way, the search marches from one triangle to the next in the general direction of the point as shown in Fig. 8. This ingenious searching algorithm is due to Lawson and avoids the need to search all of the triangles in the grid.

**Function SWAP**

This function checks to determine if a pair of adjacent triangles form a convex quadrilateral with the maximum minimum angle. The adjacent triangles share an edge V1-V2 and form a quadrilateral with vertices P-V2-V3-V1 as shown in Fig. 9. The diagonal V1-V2 is replaced with the diagonal P-V3 if P lies inside the circumcircle for the triangle V1-V2-V3. With reference to Fig. 9, we see that P lies inside the circumcircle if $2\pi - 2\gamma < 2\beta$, i.e. if $\alpha + \beta > \pi$. Similarly, P lies outside the circumcircle if $\alpha + \beta < \pi$. The neutral case occurs when $P$ lies on the circumcircle and $\pi = \alpha + \beta$. Since $\alpha + \beta < 2\pi$, a swap needs to be performed if

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

Using the formula

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

this condition is equivalent to

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

where

$$x_{13} = x_1 - x_3 \quad y_{13} = y_1 - y_3$$
$$x_{23} = x_2 - x_3 \quad y_{23} = y_2 - y_3$$
$$x_{1p} = x_1 - x_3 \quad y_{1p} = y_1 - y_3$$
$$x_{2p} = x_2 - x_3 \quad y_{2p} = y_2 - y_3$$

Thus, P lies inside the circumcircle if

$$(x_{13}x_{23} + y_{13}y_{23})(x_{13}y_{1p} - x_{1p}y_{1p} + x_{13}y_{2p} - x_{2p}y_{2p})$$
$$< (y_{13}x_{23} - x_{13}y_{23})(x_{1p}y_{1p} + y_{1p}y_{1p})$$

This check, which is due to Cline and Renka, is particularly efficient since it requires only ten multiplications, two additions and two subtractions. As pointed out by these authors, however, round-off error may cause an incorrect decision to be made when $\sin(\alpha + \beta)$ approaches zero. This condition arises when:

1. $\alpha + \beta$ is near $\pi$. 

---

**Figure 8.** Triangle searching algorithm.

**Figure 9.** Geometry for circumcircle test.
(2) \( \alpha \) and \( \beta \) are both near 0.
(3) \( \alpha \) and \( \beta \) are both near \( \pi \).

The first case occurs when \( P \) is very close to the circumcircle of an adjacent triangle, whilst the second and third cases occur when the four vertices of the adjacent triangles are nearly collinear. A complete discussion of this problem has been given by Cline and Renka. They established that the first condition has no ill-effects on the construction of a triangulation, except that the outcome of a swap test is not predictable. Allowing for the precision of floating point arithmetic, the triangulation produced will still be a Delaunay triangulation. The second and third cases, however, need to be accounted for as they may result in an incorrect triangulation. If \( \alpha \) and \( \beta \) are both in the vicinity of \( \pi \), a swap should be performed. The Cline and Renka test, which is implemented in the logical function SWAP, is as follows:

\[
\begin{align*}
\text{STEP 1:} & \quad \text{Set } \cos \alpha = \frac{x_{12}x_{23} + y_{12}y_{23}}{x_{12}^2 + y_{12}^2} \\
& \quad \text{Set } \cos \beta = \frac{x_{23}x_{31} + y_{23}y_{31}}{x_{23}^2 + y_{23}^2} \\
\text{STEP 2:} & \quad \text{If } \cos \alpha > 0 \text{ and } \cos \beta > 0 \text{ then} \\
& \quad \text{Set SWAP to FALSE and EXIT} \\
\text{STEP 3:} & \quad \text{If } \cos \alpha < 0 \text{ and } \cos \beta < 0 \text{ then} \\
& \quad \text{Set SWAP to TRUE and EXIT} \\
\text{STEP 4:} & \quad \text{Set } \sin \alpha = \frac{x_{12}y_{23} - x_{23}y_{12}}{x_{12}^2 + y_{12}^2} \\
& \quad \text{Set } \sin \beta = \frac{x_{23}y_{31} - x_{31}y_{23}}{x_{23}^2 + y_{23}^2} \\
& \quad \text{Set } \sin \alpha \cdot \cos \beta + \sin \beta \cdot \cos \alpha \\
\text{STEP 4:} & \quad \text{If } \sin \alpha \cdot \cos \beta + \sin \beta \cdot \cos \alpha < 0 \text{ then set SWAP to TRUE} \\
& \quad \text{and EXIT. Else, set SWAP to FALSE and EXIT.}
\end{align*}
\]

Although this algorithm requires a number of additional comparisons, it has been found to be relatively efficient as well as being numerically stable.

ANALYSIS OF ALGORITHM

In the bin sorting phase, the largest amount of work occurs in the quicksort procedure which requires an average of \( O(N \log_2 N) \) operations. The actual triangulation algorithm is comprised of two distinct steps; the searching step and the swapping step. If the distribution of the points throughout the \( x-y \) plane is reasonably uniform, there will be roughly \( 0(N^{3/4}) \) points in each bin. If the bins are approximately square, \( O(N^{1/4}) \) triangles will be searched to find the triangle enclosing each newly introduced point. The swapping algorithm requires roughly a constant number of operations for each point. (Empirical tests, for points distributed randomly within a square domain, indicate that an average of three swaps per point are necessary.) Once the triangulation has been completed, \( O(N) \) operations are required to delete all of the triangles that contain one or more of the supertriangle vertices. Thus, overall, the algorithm requires an average of \( O(N \log_2 N) + O(N^{3/4}) + O(N) + O(N) \) operations. For large \( N \), the average run time of the scheme is \( O(N^{5/4}) \).

As noted in a previous section, the bin sorting phase may be omitted from the algorithm if desired. The searching step will then examine roughly \( 0(N^{1/2}) \) triangles as each point is inserted and, overall, an average of \( 0(N^{3/2}) + O(N) \) + \( 0(N) \) operations are required. Thus the average run time of the algorithm without the bin sort is \( 0(N^{3/2}) \).

The worst case run time for the algorithm is \( 0(N^2) \) and occurs, for example, when the set of data points lie on a parabola. This example is discussed in detail by Lee and Schachter, and is the worst case for a variety of Delaunay triangulation schemes.

Excluding the memory required to store the \( x-y \) co-ordinates of the points and supertriangle vertices, subroutine DELTRI requires a total of \( 14N + 6 \) integer words of memory to compute and store the Delaunay triangulation. In addition to this requirement, subroutine QSORT requires 64 words of locally-declared integer memory for the operation of two stacks (this is sufficient to sort a list of \( 2^{32} \) points).

APPLICATIONS

To assess the validity and efficiency of the proposed algorithm, it was applied to sets of points distributed randomly within a unit square. Figure 10 illustrates the Delaunay triangulation for ten such points. The performance of the scheme was measured by constructing Delaunay triangulations for sets of 100, 500, 1000, 3000, 4000, 5000 and 10 000 points. The CPU times for the proposed algorithm are shown in Table 2, together with the CPU times for the implementations given by Sloan and Houlsby and Renka. These statistics are for the VAX 11/780 with full optimisation on the FORTRAN 77 compiler, and were measured using the internal clock. Two versions of the proposed algorithm were run; one with bin sorting and one without bin sorting. To assess the validity of the triangulation produced, a number of checks were conducted (the CPU times for these checks are not included in the timing statistics). Firstly, each triangle was tested to ensure that no data point lay within its circumcircle (allowing for the precision of the machine). Secondly, the area of each

Figure 10. Delaunay triangulation for ten such points
Table 2. Timing statistics for triangulation algorithms (points distributed randomly over a unit square)

<table>
<thead>
<tr>
<th>N</th>
<th>Proposed algorithm (no bin sort)</th>
<th>Proposed algorithm Cline and Renka algorithm (Renka*)</th>
<th>Watson algorithm (Sloan and Houlsby*)</th>
<th>T2/T1</th>
<th>T3/T1</th>
<th>T4/T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.36 a 1.07 a 0.36 a 1.16 a 0.41 a 1.17 a 0.37 a 1.28 a 1.00 a 1.14 a 1.03 a</td>
<td>500 2.06 a 1.01 a 2.32 a 1.21 a 2.68 a 1.31 a 2.92 a 1.26 a 1.13 a 1.30 a 1.42 a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>4.15 a 1.06 a 5.35 a 1.23 a 6.66 a 1.45 a 7.01 a 1.29 a 17.10 a 1.45 a 2.10 a 1.97 a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>8.67 a 1.10 a 12.59 a 1.32 a 18.18 a 1.52 a 28.87 a 1.59 a 2.48 a 2.13 a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>13.55 a 1.08 a 21.53 a 1.34 a 33.67 a 1.48 a 41.18 a 1.35 a 1.71 a 2.78 a 2.23 a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>18.50 a 1.03 a 31.67 a 1.29 a 51.40 a 1.57 a 55.65 a 1.81 a 3.13 a 2.39 a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>23.30 a 1.06 a 42.19 a 1.36 a 73.01 a 1.48 a 137.45 a 1.30 a 2.23 a 4.17 a 2.82 a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10000</td>
<td>48.71 a 108.48 a 203.29 a 137.45 a 2.23 a 4.17 a 2.82 a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:
1. All times in seconds for VAX 11/780.
2. 'a' values obtained by assuming times are 0(Na).
3. Sloan and Houlsby* implementation converted to single precision arithmetic.
4. Renka* implementation run with points presorted in ascending sequence of their x-co-ordinates.

The observed average case run times for the Watson and Cline and Renka schemes are in reasonable agreement with the theoretical predictions. As shown in Table 1, the average run times for these two algorithms are expected to be O(N^3/2) and O(N^4/3) respectively. Averaging the results for all values of N considered, the observed run times are approximately O(N^1.42) and O(N^1.28). The observed run times for the proposed algorithm, however, grow at a rate which is substantially less than the theoretical prediction of O(N^5/4). For values of N between 100 and 10,000, the observed run time of the proposed scheme is approximately O(N^1.66). Empirical tests indicate that the theoretical operation counts are correct. The apparent discrepancy is due to the fact that the time required for one iteration of the search procedure is substantially less than the time required for one iteration of the swapping procedure. For the values of N considered, the average number of searches conducted is small and most of the time is spent in the swapping phase. Thus, overall, the average run time of the algorithm is slightly greater than O(N) unless N is very large. Similar arguments hold when the proposed algorithm is used without the bin sort, except that the discrepancy between the predicted and observed run times is less due to the increased number of searches conducted.

CONCLUSIONS

An algorithm has been described for computing Delaunay triangulations in the plane. A FORTRAN 77 implementation of the scheme is given. Empirical tests indicate that the procedure is efficient and may be used for both small and large sets of points.

REFERENCES

APPENDIX ONE: Delaunay Triangulation Program

SUBROUTINE DELTRI(NUMPTS, N, X, Y, LIST, BIN, V, E, NUMTRI)

Purpose:
---
ASSEMBLE DELAUNAY TRIANGULATION FOR COLLECTION OF POINTS IN THE
PLANE
Input:
---
'NUMPTS' - TOTAL NUMBER OF POINTS IN DATA SET
'N' - TOTAL NUMBER OF POINTS TO BE TRIANGULATED
-N ≤ NUMPTS
'X' - X-COORDS OF ALL POINTS IN DATA SET
-x-COORD OF POINT I GIVEN BY X(I)
-LIST OF LENGTH NUMPTS+3
-LAST THREE LOCATIONS ARE USED TO STORE X-COORDS OF
SUPERTRIANGLE VERTICES IN SUBROUTINE DELAUN
'Y' - Y-COORDS OF ALL POINTS IN DATA SET
-y-COORD OF POINT I GIVEN BY Y(I)
-LIST OF LENGTH NUMPTS+3
-LAST THREE LOCATIONS ARE USED TO STORE Y-COORDS OF
SUPERTRIANGLE VERTICES IN SUBROUTINE DELAUN
'LIST' - LIST OF POINTS TO BE TRIANGULATED
-LIST OF LENGTH N
-IF N ≠ NUMPTS, SET LIST(I)=I FOR I=1,2,...,NUMPTS
PRIOR TO CALLING THIS ROUTINE
'BIN' - NOT DEFINED
-LIST OF LENGTH NUMPTS
-USED AS WORKSPACE IN SUBROUTINES BSORT AND DELAUN
'V' - NOT DEFINED
-V HAS DIMENSIONS V(3,2*N+1), WHERE N IS THE NUMBER OF
POINTS TO BE TRIANGULATED
'E' - NOT DEFINED
-E HAS DIMENSIONS E(3,2*N+1), WHERE N IS THE NUMBER OF
POINTS TO BE TRIANGULATED
'NUMTRI' - NOT DEFINED
Output:
---
'NUMPTS' - UNCHANGED
'N' - UNCHANGED
'X' - UNCHANGED, EXCEPT THAT LAST THREE LOCATIONS CONTAIN
NORMALISED X-COORDS OF SUPERTRIANGLE VERTICES
'Y' - UNCHANGED, EXCEPT THAT LAST THREE LOCATIONS CONTAIN
NORMALISED Y-COORDS OF SUPERTRIANGLE VERTICES
'LIST' - UNCHANGED

'BIN' - NOT DEFINED
'V' - VERTEX ARRAY FOR TRIANGULATION
- VERTICES LISTED IN ANTICLOCKWISE SEQUENCE
- VERTICES FOR TRIANGLE J ARE FOUND IN V(I,J) FOR I=1,2,3
  AND J=1,2,...,NUMTRI
- FIRST VERTEX IS AT POINT OF CONTACT OF FIRST AND THIRD
  ADJACENT TRIANGLES
'E' - ADJACENCY ARRAY FOR TRIANGULATION
- TRIANGLES ADJACENT TO J ARE FOUND IN E(I,J) FOR I=1,2,3
  AND J=1,2,...,NUMTRI
- ADJACENCY TRIANGLES LISTED IN ANTICLOCKWISE SEQUENCE
- ZERO DENOTES NO ADJACENT TRIANGLE
'NUMTRI' - TOTAL NUMBER OF TRIANGLES IN FINAL TRIANGULATION
- NUMTRI LT 2\*N+1

SUBROUTINES CALLED:
---------------------
BSORT,DELAUN

PROGRAMMER:
------------
S W SLOAN

LAST MODIFIED:
---------------
30 JAN 1986  S W SLOAN

*******************************************************************************

INTEGER N,I,LIST(*),V(3,*),E(3,*),NUMTRI,BIN(*),P,NUMPTS
REAL XMIN,XMAX,YMIN,YMAX,DMAX,COOOO1,FACT,X(*),Y(*)
PARAMETER(COO001=1.0)

COMPUTE MIN AND MAX COORDS FOR X AND Y
COMPUTE MAX OVERALL DIMENSION
   XMIN=X(LIST(1))
   XMAX=XMIN
   YMIN=Y(LIST(1))
   YMAX=YMIN
   DO 5 I=2,N
       P=LIST(I)
       XMIN=MIN(XMIN,X(P))
       XMAX=MAX(XMAX,X(P))
       YMIN=MIN(YMIN,Y(P))
       YMAX=MAX(YMAX,Y(P))
   5 CONTINUE
   DMAX=MAX(XMAX-XMIN,YMAX-YMIN)

NORMALISE X-Y COORDS OF POINTS
   FACT=COOOO1/DMAX
   DO 10 I=1,N
       P=LIST(I)
       X(P)=(X(P)-XMIN)*FACT
       Y(P)=(Y(P)-YMIN)*FACT
   10 CONTINUE

SORT POINTS INTO BINS
THIS CALL IS OPTIONAL
   CALL BSORT(N,X,Y,XMIN,XMAX,YMIN,YMAX,DMAX,BIN,LIST)

COMPUTE DELAUNAY TRIANGULATION
   CALL DELAUN(NUMPTS,N,X,Y,LIST,BIN,V,E,NUMTRI)

RESET X-Y COORDS TO ORIGINAL VALUES
   DO 30 I=1,N
       P=LIST(I)
       X(P)=X(P)*DMAX+XMIN
       Y(P)=Y(P)*DMAX+YMIN
   30 CONTINUE

END
SUBROUTINE BSORT(N,X,XMIN,XMAX,YMIN,YMAX,DMAX,BIN,LIST)

PURPOSE:
--------
SORT POINTS SUCH CONSECUTIVE POINTS ARE CLOSE TO ONE ANOTHER IN THE
X-Y PLANE USING A BIN SORT

INPUT:
-------
'N' - TOTAL NUMBER OF POINTS TO BE TRIANGULATED
     - N LE NUMPTS, WHERE NUMPTS IS TOTAL NUMBER OF POINTS IN
       DATA SET
'X' - X-COORDS OF ALL POINTS IN DATA SET
     - IF POINT IS IN LIST, THE COORDINATE MUST BE NORMALISED
       ACCORDING TO X=(X-XMIN)/DMAX
     - X-COORD OF POINT I GIVEN BY X(I)
     - LIST OF LENGTH NUMPTS+3
     - LAST THREE LOCATIONS ARE USED TO STORE X-COORDS OF
       SUPERTRIANGLE VERTICES IN SUBROUTINE DELAUN
'Y' - Y-COORDS OF ALL POINTS IN DATA SET
     - IF POINT IS IN LIST, THE COORDINATE MUST BE NORMALISED
       ACCORDING TO Y=(Y-YMIN)/DMAX
     - Y-COORD OF POINT I GIVEN BY Y(I)
     - LIST OF LENGTH NUMPTS+3
     - LAST THREE LOCATIONS ARE USED TO STORE Y-COORDS OF
       SUPERTRIANGLE VERTICES IN SUBROUTINE DELAUN
'XMIN' - MIN X-COORD OF POINTS IN LIST
'XMAX' - MAX X-COORD OF POINTS IN LIST
'YMIN' - MIN Y-COORD OF POINTS IN LIST
'YMAX' - MAX Y-COORD OF POINTS IN LIST
'DMAX' - DMAX=MAX(XMAX-XMIN,YMAX-YMIN)
'BIN' - NOT DEFINED
     - LIST OF LENGTH NUMPTS
'LIST' - LIST OF POINTS TO BE TRIANGULATED
     - LIST OF LENGTH N

OUTPUT:
-------
'N' - UNCHANGED
'X' - UNCHANGED
'Y' - UNCHANGED
'XMIN' - UNCHANGED
'XMAX' - UNCHANGED
'YMIN' - UNCHANGED
'YMAX' - UNCHANGED
'DMAX' - UNCHANGED
'BIN' - BIN NUMBERS FOR EACH POINT TO BE TRIANGULATED
     - LIST OF LENGTH NUMPTS
'LIST' - LIST OF POINTS TO BE TRIANGULATED
     - POINTS ORDERED SUCH THAT CONSECUTIVE POINTS ARE CLOSE
       TO ONE ANOTHER IN THE X-Y PLANE

SUBROUTINES CALLED:
---------------------
QSORTI

PROGRAMMER:
-----------
S W SLOAN

LAST MODIFIED:
---------------
30 JAN 1986 S W SLOAN

* INTEGER LIST(*),BIN(*),N,I,J,K,P,NDIV

* REAL X(*),Y(*),FACTX,FACTY,XMIN,XMAX,YMIN,YMAX,DMAX

* COMPUTE NUMBER OF BINS IN X-Y DIRECTIONS
* COMPUTE INVERSE OF BIN SIZE IN X-Y DIRECTIONS
* NDIV=NINT(REAL(N)**0.25)
* FACTX=REAL(NDIV)/((XMAX-XMIN)**1.0I/DMAX)
* FACTY=REAL(NDIV)/((YMAX-YMIN)**1.0I/DMAX)

* ASSIGN BIN NUMBERS TO EACH POINT
* DO
  10 K=1,N
  I=INT(Y(P)*FACTY)
  J=INT(X(P)*FACTX)
  IF(MOD(I,2).EQ.0)THEN
    BIN(P)=I*NDIV+J
  ELSE
    BIN(P)=(I+1)*NDIV-J
  END IF
  10 CONTINUE

* SORT POINTS IN ASCENDING SEQUENCE OF BIN NUMBER
* CALL QSORTI(N,LIST,BIN)
* END

SUBROUTINE QSORTI(N,LIST,KEY)

* PURPOSE:
* ORDER LIST OF INTEGERS IN ASCENDING SEQUENCE OF THEIR INTEGER KEYS
* INPUT:
* 'N' - POSITIVE INTEGER GIVING LENGTH OF LIST
* 'LIST' - LIST OF INTEGERS TO BE SORTED
* 'KEY' - LIST OF INTEGER KEYS
* OUTPUT:
* 'N' - UNCHANGED
* 'LIST' - LIST OF INTEGERS SORTED IN ASCENDING SEQUENCE OF THEIR KEYS
* 'KEY' - UNCHANGED

* NOTES:
* - USES QUICKSORT ALGORITHM, EFFICIENT FOR 'N' VALUES GREATER THAN
  ABOUT 12 (ALTHOUGH MAY BE SYSTEM DEPENDENT)
  - ROUTINE SortS Lists UPTo LENGTH 2**MAXSTK

PROGRAMMER:

INTEGER LIST(*), KEY(*), N, LL, LR, LM, NL, NR, LTEMP, STKTOP, MAXSTK, GUESS
PARAMETER(MAXSTK=32)
INTEGER LSTACK(MAXSTK), RSTACK(MAXSTK)

LL = 1
LR = N
STKTOP = 0
10 IF(LL.LT.LR) THEN
   NL = LL
   NR = LR
   LM = (LL + LR)/2
   GUESS = KEY(LIST(LM))
   CALL FIND KEYS FOR EXCHANGE
20 IF(KEY(LIST(NL)).LT.GUESS) THEN
   NL = NL + 1
   GOTO 20
END IF
30 IF(GUESS.LT.KEY(LIST(NR))) THEN
   NR = NR - 1
   GOTO 30
END IF
IF(NL.LT.(NR - 1)) THEN
   LTEMP = LIST(NL)
   LIST(LH) = LIST(NR)
   LIST(NR) = LTEMP
   NL = NL + 1
   NR = NR - 1
   GOTO 20
END IF
CALL DEAL WITH CROSSING OF POINTERS
IF(NL.LE.NR) THEN
   IF(NL.LT.NR) THEN
      LTEMP = LIST(NL)
      LIST(NL) = LIST(NR)
      LIST(NR) = LTEMP
   END IF
   NL = NL + 1
   NR = NR - 1
END IF
CALL SELECT SUB-LIST TO BE PROCESSED NEXT
STKTOP = STKTOP + 1
IF(NR.LT.LH) THEN
   LSTACK(STKTOP) = NL
   RSTACK(STKTOP) = LR
   LH = NR
ELSE
   LSTACK(STKTOP) = LL
   RSTACK(STKTOP) = NR
   LL = NL
END IF
GOTO 10
END IF
CALL PROCESS ANY STACKED SUB-LISTS
IF(STKTOP.NE.0) THEN
   LL = LSTACK(STKTOP)
   LR = RSTACK(STKTOP)
   STKTOP = STKTOP - 1
   GOTO 10
END IF
END
SUBROUTINE DELAUN(NUMPTS,N,X,Y,LIST,STACK,V,E,NUMTRI)

SUBROUTINE DELAUN

PURPOSE:
-------

ASSEMBLE DELAUNAY TRIANGULATION

INPUT:
-------

'NUMPTS' - TOTAL NUMBER OF POINTS IN DATA SET

'N' - TOTAL NUMBER OF POINTS TO BE TRIANGULATED

'X' - X-COORDS OF ALL POINTS IN DATA SET

'Y' - Y-COORDS OF ALL POINTS IN DATA SET

'LIST' - LIST OF POINTS TO BE TRIANGULATED

'STACK' - NOT DEFINED

'V' - VERTEX ARRAY FOR TRIANGULATION

'E' - ADJACENCY ARRAY FOR TRIANGULATION

'NUMTRI' - NUMBER OF TRIANGLES IN FINAL TRIANGULATION

OUTPUT:
-------

'NUMPTS' - UNCHANGED

'N' - UNCHANGED

'X' - UNCHANGED

'Y' - UNCHANGED

'LIST' - UNCHANGED

'STACK' - NOT DEFINED

'V' - VERTEX ARRAY FOR TRIANGULATION

'E' - ADJACENCY ARRAY FOR TRIANGULATION

'NUMTRI' - NUMBER OF TRIANGLES IN FINAL TRIANGULATION
SUBROUTINES CALLED:

-------------------
PUSH

FUNCTIONS CALLED:

-------------
TRILOC, EDC, SWAP, POP

PROGRAMMER:

--------
S W SLOAN

LAST MODIFIED:

---------------------
30 JAN 1986 S W SLOAN

******************************************************************************

INTEGER V(3, *), N, I, LIST(*), NUMTRI, P, E(3, *), MAXSTK, TOPSTK,
+ V1, V2, V3, L, R, POP, A, B, C, ERL, ERA, ERB, EDC, TRILOC, NUMPTS,
+ TSTRT, TSTOP, STACK(*)

REAL X(*), Y(*), XP, YP, CO0000, CO0100

LOGICAL SWAP

PARAMETER (CO0000=0.0,
+ CO0100=100.0)

DEFINE VERTEX AND ADJACENCY LISTS FOR SUPERTRIANGLE

V1=NUMPTS+1
V2=NUMPTS+2
V3=NUMPTS+3
V(1,1)=V1
V(2,1)=V2
V(3,1)=V3
E(1,1)=0
E(2,1)=0
E(3,1)=0

SET COORDS OF SUPERTRIANGLE

X(V1)=-CO0100
X(V2)=CO0100
X(V3)=CO0000
Y(V1)=-CO0100
Y(V2)=-CO0100
Y(V3)=CO0100

LOOP OVER EACH POINT

NUMTRI=1
TOPSTK=0
MAXSTK=NUMPTS
DO 100 I=1, N
  P=LIST(I)
  XP=X(P)
  YP=Y(P)
  T=TRILOC(XP, YP, X, X, V, E, W, D, MTRI)
  CREATE NTRI VERTEX AND ADJACENCY LISTS FOR TRIANGLE T
  A=E(1, T)
  B=E(2, T)
  C=E(3, T)
  V1=V(A, T)
  V2=V(B, T)
  V3=V(C, T)
  V1, T)=P
  V(2, T)=V1
  V(3, T)=V2
  E(1, T)=NUMTRI+2
  E(2, T)=A
  E(3, T)=NUMTRI+1

CREATE NEW TRIANGLES
NUMTRI = NUMTRI + 1
V(1, NUMTRI) = P
V(2, NUMTRI) = V2
V(3, NUMTRI) = V3
E(1, NUMTRI) = T
E(2, NUMTRI) = B
E(3, NUMTRI) = NUMTRI + 1
NUMTRI = NUMTRI + 1
V(1, NUMTRI) = P
V(2, NUMTRI) = V2
V(3, NUMTRI) = V3
E(1, NUMTRI) = T
E(2, NUMTRI) = C
E(3, NUMTRI) = T

PUT EACH EDGE OF TRIANGLE T ON STACK
STORE TRIANGLES ON LEFT SIDE OF EACH EDGE
UPDATE ADJACENCY LISTS FOR ADJACENT TRIANGLES
ADJACENCY LIST FOR ELEMENT A DOES NOT NEED TO BE UPDATED

IF(A .NE. 0) THEN
    CALL PUSH(T, MAXSTK, TOPSTK, STACK)
END IF
IF(B .NE. 0) THEN
    E(EDG(B, T, E), B) = NUMTRI - 1
    CALL PUSH(NUMTRI - 1, MAXSTK, TOPSTK, STACK)
END IF
IF(C .NE. 0) THEN
    E(EDG(C, T, E), C) = NUMTRI
    CALL PUSH(NUMTRI, MAXSTK, TOPSTK, STACK)
END IF
LOOP WHILE STACK IS NOT EMPTY

IF(TOPSTK .GT. 0) THEN
    L = POP(TOPSTK, STACK)
    R = E(2, L)
    CHECK IF NEW POINT IS IN CIRCUMCIRCLE FOR TRIANGLE R
    ERL = EDG(R, L, E)
    ERA = MOD(ERL, 3) + 1
    ERE = MOD(ERA, 3) + 1
    V1 = V(ERL, R)
    V2 = V(ERA, R)
    V3 = V(ERE, R)
    IF(SWAP(X(V1), Y(V1), X(V2), Y(V2), X(V3), Y(V3), XP, YP)) THEN
        NEW POINT IS INSIDE CIRCUMCIRCLE FOR TRIANGLE R
        SWAP DIAGONAL FOR CONVEX QUAD FORMED BY P-V2-V3-V1
        A = E(ERA, R)
        B = E(ERE, R)
        C = B(3, L)
        UPDATE VERTEX AND ADJACENCY LIST FOR TRIANGLE L
        V(3, L) = V3
        E(2, L) = A
        E(3, L) = R
        UPDATE VERTEX AND ADJACENCY LIST FOR TRIANGLE R
        V(1, R) = P
        V(2, R) = V3
        V(3, R) = V1
        E(1, R) = L
        E(2, R) = B
        E(3, R) = C
        PUT EDGES L-A AND R-B ON STACK
        UPDATE ADJACENCY LISTS FOR TRIANGLES A AND C
        IF(A .NE. 0) THEN
            E(EDG(A, R, E), A) = L
            CALL PUSH(L, MAXSTK, TOPSTK, STACK)
        END IF
        IF(B .NE. 0) THEN
            CALL PUSH(R, MAXSTK, TOPSTK, STACK)
        END IF
        IF(C .NE. 0) THEN
            E(EDG(C, L, E), C) = R
        END IF
    END IF
END IF
GOTO 50
END IF
100 CONTINUE

CHECK CONSISTENCY OF TRIANGULATION

IF(NUMTRI.NE.2*N+1) THEN
  WRITE(6,'("***ERROR IN SUBROUTINE DELAUNAY")')
  WRITE(6,'("***INCORRECT NUMBER OF TRIANGLES FORMED")')
  STOP
END IF

REMOVE ALL TRIANGLES CONTAINING SUPERTRIANGLE VERTICES.

FIND FIRST TRIANGLE TO BE DELETED (TRIANGLE T)

UPDATE ADJACENCY LISTS FOR TRIANGLES ADJACENT TO T

DO 120 T=1,NUMTRI
  IF((V(1,T).GT.NUMPTS).OR.
     + (V(2,T).GT.NUMPTS).OR.
     + (V(3,T).GT.NUMPTS)) THEN
    DO 110 I=1,3
      A=E(I,T)
      IF(A.NE.0) THEN
        E(EDG(A,T,E),A)=0
      END IF
    110 CONTINUE
    GOTO 125
  END IF
120 CONTINUE

REMOVE TRIANGLES

DO 200 T=TSTRT,TSTOP
  IF((V(1,T).GT.NUMPTS).OR.
     + (V(2,T).GT.NUMPTS).OR.
     + (V(3,T).GT.NUMPTS)) THEN
    TRIANGLE T IS TO BE DELETED
    UPDATE ADJACENCY LISTS FOR TRIANGLES ADJACENT TO T
    DO 130 I=1,3
      A=E(I,T)
      IF(A.NE.0) THEN
        E(EDG(A,T,E),A)=0
      END IF
    130 CONTINUE
  ELSE
    TRIANGLE T IS NOT TO BE DELETED
    PUT TRIANGLE T IN PLACE OF TRIANGLE NVTMTRI
    UPDATE ADJACENCY LISTS FOR TRIANGLES ADJACENT TO T
    NUMTRI=NUMTRI+1
    DO 140 I=1,3
      V(I,NUMTRI)=V(I,T)
    140 CONTINUE
  END IF
200 CONTINUE
END

SUBROUTINE PUSH (ITEM, MAXSTK, TOPSTK, STACK)

SUBROUTINE PUSH

PURPOSE:

PLACE ITEM ON LIFO STACK AND INCREMENT STACK SIZE

INPUT:

'ITEM' - ITEM TO BE PLACED AT TOP OF LIFO STACK
'MAXSTK' - MAX SIZE OF STACK
'TOPSTK' - POINTER INDICATING CURRENT SIZE OF STACK
  - MUST BE LT MAXSTK WHEN THIS ROUTINE IS CALLED
'STACK' - LIFO STACK

OUTPUT:
-------
'ITEM' - UNCHANGED
'MAXSTK' - UNCHANGED
'TOPSTK' - POINTER INDICATING CURRENT SIZE OF STACK
  - NEW VALUE = OLD VALUE + 1
'STACK' - LIFO STACK WITH ITEM ADDED
  - STACK(TOPSTK)=ITEM

PROGRAMMER:
-------------
S W SLOAN

LAST MODIFIED:
---------------
30 JAN 1986 S W SLOAN

******************************************************************************************
INTEGER TOPSTK,MAXSTK,STACK(*),ITEM

TOPSTK=TOPSTK+1
IF(TOPSTK.GT.MAXSTK)THEN
  WRITE(6,'(Err.Error in Subroutine Push)')
  WRITE(6,'(***Stack overflow)')
  STOP
ELSE
  STACK(TOPSTK)=ITEM
END IF
END

FUNCTION POP(TOPSTK,STACK)

FUNCTION POP
PURPOSE:
--------
REMOVE ITEM FROM LIFO STACK AND DECREMENT STACK SIZE

INPUT:
-------
'TOPSTK' - POINTER INDICATING CURRENT SIZE OF STACK
  - MUST BE GT ZERO WHEN THIS FUNCTION IS CALLED
'STACK' - LIFO STACK
'POP' - NOT DEFINED

OUTPUT:
-------
'TOPSTK' - POINTER INDICATING SIZE OF STACK
  - NEW VALUE = OLD VALUE - 1
'STACK' - UNCHANGED
'POP' - ITEM AT TOP OF STACK WHEN FUNCTION WAS CALLED

PROGRAMMER:
-------------
FUNCTION EDG(L,K,E)

PURPOSE:
----------
FIND EDGE IN TRIANGLE L WHICH IS ADJACENT TO TRIANGLE K

INPUT:
------

' L ' - NUMBER OF TRIANGLE

' K ' - NUMBER OF ADJACENT TRIANGLE

' E ' - ADJACENCY ARRAY FOR TRIANGULATION

OUTPUT:
-------

' L ' - UNCHANGED

' K ' - ADJACENT TRIANGLES LISTED IN ANTICLOCKWISE SEQUENCE

' E ' - ADJACENT TRIANGLES LISTED IN ANTICLOCKWISE SEQUENCE

' EDG ' - NUMBER OF EDGE IN TRIANGLE L WHICH IS ADJACENT TO

TRIANGLE K

' EDG ' = E(EDG, L) = K

PROGRAMMER:
------------
S W SLOAN

LAST MODIFIED:
-----------
30 JAN 1986 S W SLOAN

INTEGER L,K,I,E(3,*),EDG

DO 10 I=1,3
  IF(E(I,L).EQ.K)THEN
    EDG=I
    RETURN
  END IF
10 CONTINUE
FUNCTION TRILOC(XP,YP,X,Y,V,E,NUMTRI)

PROGRAMMER:
S W SLOAN
LAST MODIFIED:
30 JAN 1986  S W SLOAN

INTEGER V(3,*),E(3,*),NUMTRI,V1,V2,I,T,TRILOC
REAL X(*),Y(*),XP,YP

T=NUMTRI
10 CONTINUE
DO 20 I=1,3
V1=V(I,T)
V2=V(MOD(I,3)+1,T)

IF((Y(V1)-YP)*(X(V2)-XP).GT.(X(V1)-XP)*(Y(V2)-YP))THEN
  T=E(I,T)
  GOTO 10
END IF

TRIANGLE HAS BEEN FOUND
TRILOC=T
END

FUNCTION SWAP(X1,Y1,X2,Y2,X3,Y3,XP,YP)

PROGRAMMER:  S W Sloan
LAST MODIFIED:  30 JAN 1986 S W Sloan

REAL X1,Y1,X2,Y2,X3,Y3,XP,YP,X13,Y13,X23,Y23,X1P,Y1P,X2P,Y2P,COSA, +
  COSB,SINA,SINE,CO0000
LOGICAL SWAP
PARAMETER(CO0000=0.0)

X13=X1-X3
Y13=Y1-Y3
X23=X2-X3
Y23=Y2-Y3
X1P=X1-XP
Y1P=Y1-YP
X2P=X2-XP
Y2P=Y2-YP
COSA=X13*X23+Y13*Y23
COSB=X2P*X1P+Y2P*YP
IF((COSA.GE.CO0000).AND.(COSB.GE.CO0000))THEN
  SWAP=.FALSE.
ELSEIF((COSA.LT.CO0000).AND.(COSB.LT.CO0000))THEN
  SWAP=.TRUE.
ELSE

ENDIF

ENDIF

END

ENDIF

END

ELSE

SWAP=.FALSE.

ENDIF

END IF

ENDIF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.

ENDIF

END IF

ELSE

SWAP=.FALSE.