Computing Dirichlet tessellations†

A. Bowyer

School of Mathematics, University of Bath, Claverton Down, Bath BA2 7AY

An efficient algorithm is proposed for computing the Dirichlet tessellation and Delaunay triangulation in a k dimensional Euclidean space ($k \geqslant 2$). The algorithm is designed in a way that should allow it to be extended to some of the simpler non-Euclidean metric spaces as well. The algorithm has been implemented in ISO FORTRAN by the author and execution times and stereoscopic pictures of the tessellation and triangulation are presented at the end of this paper.

(Received April 1980)

1. Introduction

1.1 *Definitions*

Suppose the positions of n distinct points $P_1 \ldots P_n$ in the plane are given as data. We may give each point a territory that is that area of the plane nearer to it than to any other data point. The resulting territories will form a pattern of packed convex polygons covering the whole plane. This construct is known as the Dirichlet tessellation of the points. **Fig. 1** (from Green and Sibson, 1978) shows it (bold lines) for a small set of points ($n = 12$). Some data points (those on the convex hull) will have infinite territories, the rest will have territories that are finite.

From the definition above the straight line segments that form the territorial boundaries must lie half way between the two points on either side of those whose territories they help to delineate. Each segment of territorial boundary is part of the perpendicular bisector of the line joining the point pair between whose territories the boundary lies. If all point pairs which have some segment of boundary in common are joined by straight lines the result is a triangulation of the convex hull of the data points. This triangulation is known as the Delaunay triangulation. In Fig. 1 the Delaunay triangulation is shown by the faint lines. Point pairs joined by lines in the Delaunay triangulation are said to be contiguous. Green and Sibson (1978) have published an efficient [$0(n^{3/2})$; can be reduced to $0(n.\log n)$] algorithm for computing these structures in two dimensions.

The definitions given above apply more generally to Euclidean space in any number of dimensions, and it is to the problem of computing the Dirichlet tessellation and Delaunay triangulation in any Euclidean space that this paper is addressed. With care, it should also be possible to apply the algorithm for the solution of this problem to some of the better behaved non-Euclidean metric spaces as well (the surface of a sphere, for example).

Brown (1979) mentions the possibility of computing tessellations in k dimensions using transformations of convex hulls in $k + 1$ dimensions. In the absence of a general convex hull algorithm, however, it is difficult to see how this could be implemented. Indeed it may be possible to perform the reverse operation: computing convex hulls from a k dimensional Dirichlet tessellation.

1.2 *Properties*

In two dimensions the vertices of the territories occur where three territorial boundaries meet (except in degenerate cases, see Section 3.1). The three territories around a vertex belong to three points that form a Delaunay triangle. From the definition of the tessellation a vertex in it must be equidistant from all three of its forming points. It is the circumcentre of their Delaunay triangle. Each Delaunay triangle will have associated with it a unique vertex in the tessellation and vice versa.

In a k dimensional Euclidean space the Delaunay triangles become simplexes with $k + 1$ data points as vertices. Each vertex in the tessellation is where $k + 1$ territories meet and is the centre of the hypersphere passing through all the vertices of the associated simplex. As before each contiguous pair of points is joined by a line that is an edge of some Delaunay simplexes. The territorial boundary shared by the contiguous point pair is a convex polygon lying in the $k - 1$ dimensional hyperplane that bisects that edge.

In three dimensions the territory of each data point is a convex polyhedron: the region of space nearer to the point than to any other. The faces of the polyhedra will be convex polygons: the

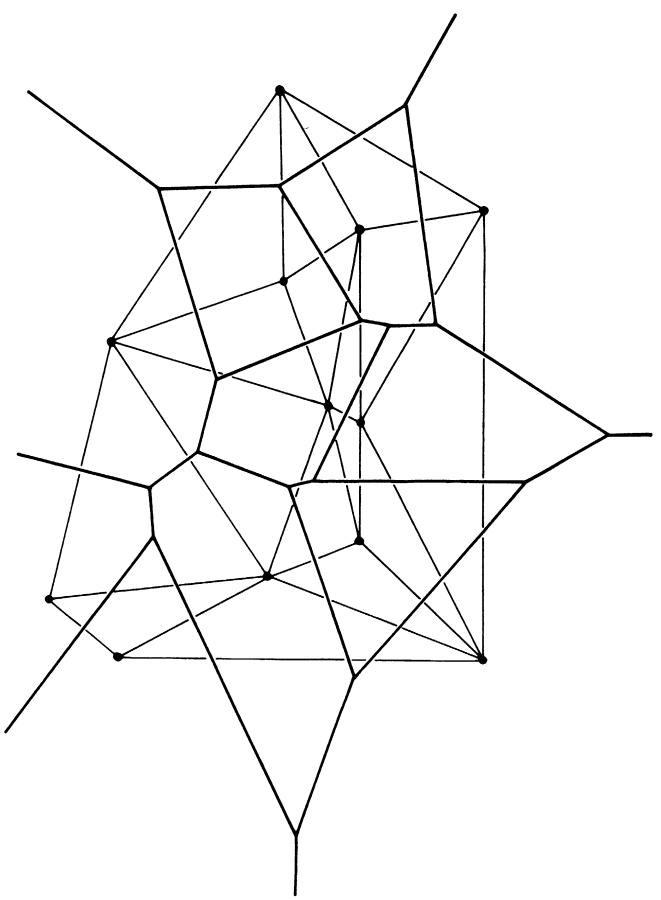

Fig. 1 The Dirichlet tessellation (bold lines) and Delaunay triangulation (fine lines) for a small-scale configuration, from Green and Sibson (1978)

†Editorial note: This paper and that by Watson (this issue) cover some material in common. As these contributions were received at approximately the same time, the Editor feels it only right to include both papers.

CCC-0010-4620/81/0024-0162 $02.50

territorial boundaries shared by contiguous points. Each convex polygon will lie in the plane that bisects an edge of a Delaunay tetrahedron. **Fig. 2** shows a three dimensional vertex and its associated Delaunay tetrahedron.

1.3 *Applications*

The availability over the past few years of an efficient two dimensional algorithm for calculating the Dirichlet tessellation has meant that it is now possible to investigate a number of applications for the tessellation that predate the algorithm, and has also opened up some new applications. Most of these are not specifically two dimensional problems.

Green (pers. com.) has used the structure to model a random pattern of cells through which an epidemic is allowed to spread. A good deal is known about the spread of epidemics on regular lattices. The Delaunay triangulation and Dirichlet tessellation of some realisations of random point processes provide a convenient random lattice for these studies with a number of useful properties.

An obvious application of the two dimensional structure is to use it to investigate the territorial behaviour of animals.

On a more abstract level the structures are useful in the analysis and simulation of certain spatial point processes (see Miles, 1970; Ripley, 1977). One of the models of crystal growth used by crystallographers calls for the use of the three dimensional version of the Dirichlet tessellation (Gilbert, 1962). Geographers use the structures for a variety of purposes.

Lawson (1972) gives an optimality criterion for triangulations that are to be used for interpolation and finite element work. Sibson (1978) has shown that the Delaunay triangulation uniquely satisfies that criterion.

Perhaps one of the most important applications of the Dirichlet tessellation is in the fitting of surfaces to, and in the smoothing of, observations of some function (barometric pressures or height above sea level, for example) taken on an irregular pattern of observation sites (Sibson, 1980). The availability of a tessellation/triangulation algorithm which will work in higher dimensional spaces will allow this work to be extended to cover numbers of independent variables greater than two.

As the algorithm presented in this paper is designed for handling convex polygons in any number of dimensions it may well prove useful in some linear programming problems. The algorithm may also be of some use in the computation of convex hulls of sets of data points in more than two dimensions (Green and Silverman, 1979; Brown, 1979).

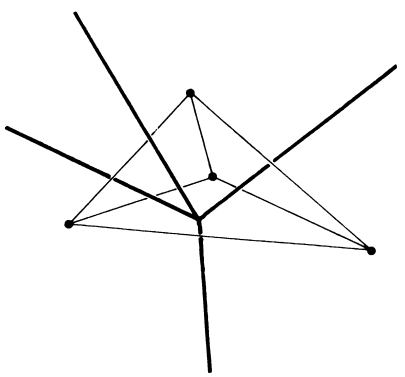

Fig. 2 3 dimensional Delaunay tetrahedron and its associated vertex

2. The algorithm

2.1 *Data structure*

Before considering how the structure may be computed a decision must be made on how to store it. Green and Sibson store the Delaunay triangulation in the form of lists of contiguous points for each point. Their algorithm takes advantage of the fact that these lists may be stored cyclically for the two dimensional case in order to facilitate the insertion of points into the structure one by one. In higher dimensions it is not possible to order the contiguity lists cyclically, though it might be possible to organise these lists in some other way (in order of increasing distance from the point of which the list entries were neighbours, for example).

Suppose it were desired to store the vertex structure of the tessellation. Consider **Fig. 3.** Here eight data points in the plane give rise to seven vertices, $V_1 \ldots V_7$. The territorial boundaries that extend to infinity can conveniently be considered as terminating in a vertex labelled zero. Each vertex is the circumcentre of three data points and a list of those points could be recorded for each vertex. In addition each vertex points to three other vertices, each one opposite one of the vertex's

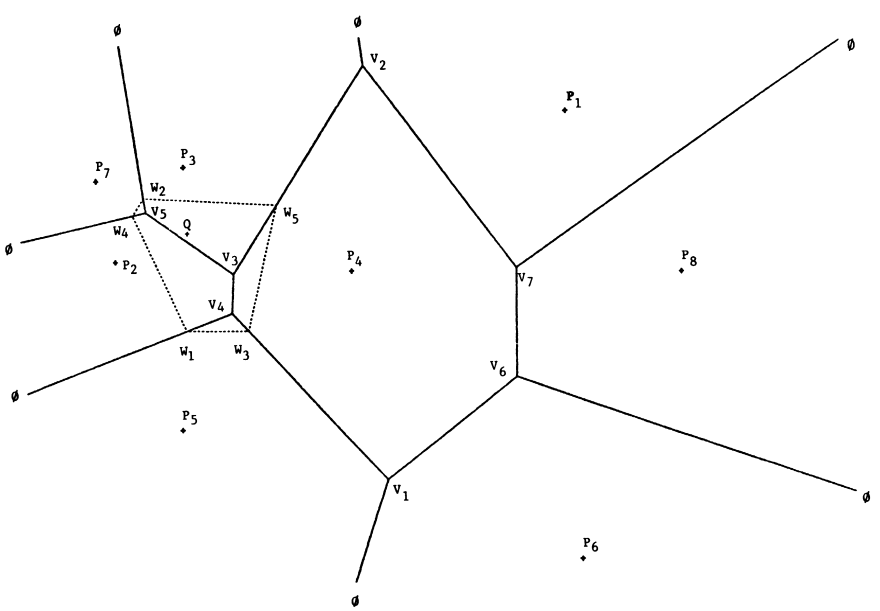

Fig. 3 The algorithm for finding the territory of point Q

Table 1

Vertex	*forming points*			*neighbouring vertices*		
	1	2	3	1	2	3
V_1	P_6	P_4	P_5	V_4	$\emptyset$	V_6
V_2	P_1	P_4	P_3	V_3	$\emptyset$	V_7
V_3	P_2	P_3	P_4	V_2	V_4	V_5
V_4	P_2	P_5	P_4	V_1	V_3	$\emptyset$
V_5	P_7	P_3	P_2	V_3	$\emptyset$	$\emptyset$
V_6	P_6	P_8	P_4	V_7	V_1	$\emptyset$
V_7	P_1	P_8	P_4	V_6	V_2	$\emptyset$

forming points. It is thus possible to record the structure by constructing two lists, each of length three, for each vertex in the structure, one list holding the forming points of the vertex (Delaunay triangles), the other holding the opposite neighbouring vertices (**Table 1**). The sense of the cyclic order of the points round a vertex is of no consequence in the algorithm to be described, and it is deliberately arbitrary in Table 1.

In k dimensions each vertex will have $k + 1$ forming points and $k + 1$ neighbouring vertices opposite them.

2.2 *Adding a point*

If it is possible to record the structure in the manner outlined above and then to add a new data point and modify the record appropriately any number of points can be tessellated and triangulated by starting with a simple structure and building upon it. The obvious starting pattern is the Delaunay simplex formed by the first $k + 1$ points. This will give a tessellation containing one real vertex all of whose neighbouring vertices will be $\emptyset$. The only restriction here is that the first $k + 1$ points must not all lie in a hyperplane in the k dimensional space that is being considered. This should not present a serious limitation.

Suppose we wish to insert a new point (Q in Fig. 3) within the current convex hull of the data points. The territory we wish to find is indicated by the dotted lines. The algorithm for doing this can be outlined as follows:

1. Identify a vertex currently in the structure that will be deleted by the new point (say V_4). Such a vertex is any that is *nearer to the new point than to its forming points*. There will always be at least one such vertex, as the vertex corresponding to the Delaunay simplex in which the new point lies will always be deleted and the Delaunay simplexes completely fill the convex hull of the currently included points.
2. Perform a tree search through the vertex structure starting at the deleted vertex looking for others that will be deleted. This is an easy matter if the data are stored as indicated in Table 1. The result will be a list of all the vertices deleted by the new point, Q. In this case the list will be: $\{V_4, V_3, V_5\}$.
3. The points contiguous to Q are all the points forming the deleted vertices: $\{P_2, P_5, P_4, P_3, P_7\}$.
4. An old contiguity between a pair of those points will be removed ($P_2 - P_4$ say) if all its vertices $\{V_4, V_3\}$ are in the list of deleted vertices.
5. In this case the new point has five new vertices associated with it: $\{W_1, W_2, W_3, W_4, W_5\}$. Compute their forming points and neighbouring vertices. The forming points for each will be the point Q and k of the points contiguous to Q. Each line in the tessellation has k points around it (the line $V_3 - V_2$, for example, is formed by P_3 and P_4). The forming points of the new vertices and their neighbouring vertices may be found by considering vertices pointed to by members of the deleted vertex list that are not themselves deleted, and finding the rings of points around them. Thus W_5 points outwards to V_2 from Q and is formed by $\{P_3, P_4, Q\}$.
6. The final step is to copy some of the new vertices, overwriting the entries of those deleted to save space.

Note that, with the exception of step one, all these operations are of a local nature and may be carried out in a time independent of the number of points currently in the structure. The amount of work to be done will be roughly proportional to the number of new vertices created. Given k the expected number of vertices in a point's territorial boundary will be constant. In two dimensions the Euler-Poincaré formula for faces, edges and vertices on a solid can be used to prove that the expected number of vertices per territory is exactly six. In three dimensions Miles (1970) gives an upper value for the expected number of vertices per territory as 27·07 for a Poisson point process. Unfortunately there is, as yet, no general expression for this number in k dimensions.

Thus, given k and step 1, the amount of work done in inserting a point will be constant, leading to an $0(n)$ term in the time to compute the structure for n points. How may a vertex that will be deleted be identified for step 1?

Clearly it would be possible to examine each vertex in the structure to see if it was nearer to the new point than to its forming points. However, this would be a time consuming process (especially with large numbers of points in many dimensions), and would remove the advantage obtained by the local nature of the insertion algorithm.

It would be an advantage to be able to identify a deleted vertex without the need to examine most of the vertices in the structure. How may this be done? Green and Sibson's algorithm for the two dimensional case starts by finding the nearest neighbouring point in the current structure to the point that is about to be inserted, that is the currently accepted point in whose territory the new point lies. They find this nearest neighbour by performing a walk from neighbour to neighbour across the Delaunay triangulation from some already accepted point towards the new point. In the absence of any other information the obvious place to start this walk is at a point near to the centroid of the currently accepted points; in two dimensions the walk would then take $0(n^{1/2})$ time. Clearly, if it were known that the new point was likely to be near the one that had just been inserted, then that last point would be the obvious place to start the walk, leading to a negligible computational load.

In k dimensions a walk starting from the centroid of the configuration should take $0(n^{1/k})$ time.

Once the nearest neighbour of the new point has been found it is a simple matter to find a deleted vertex. The new point must delete at least one vertex on the boundary of its nearest neighbour's territory. One aspect of this method of finding a deleted vertex is that it is necessary to hold lists of contiguous points for each point in the structure so that the walk may be performed easily. This may well be useful information to have available when the algorithm has finished and the lists are easy to compute and maintain (steps 3 and 4). It would be nice if it were possible to perform a walk through the vertex structure as stored in Table 1 to find a deleted vertex, thus removing the need to maintain separate lists of contiguities. It is a simple matter to walk through the vertex structure starting at some vertex near the centroid of the configuration to find the vertex nearest to the new point. Unfortunately, this vertex is not necessarily deleted by the new point and the author has been unable to devise a simple rule for finding one nearby that is.

Finally, at the beginning of the outline of the algorithm it was stated that it was for inserting a point within the current convex hull of the points. The reason for this is that a new point outside the convex hull may not delete any vertices and thus has to be treated differently. It is easy to flag when this occurs as none

of the vertices of the new point's nearest neighbour are deleted, and this is easily detected. The simplest method of getting round this difficulty is to set up the initial simplex and vertex on which the algorithm builds such that the $k + 1$ points on the corners of the simplex remain the convex hull throughout the entire process. As the whole range of floating point numbers is available this is not difficult, even for the most unusual data. These first $k + 1$ points would, almost always, not be data values, but would be artificially generated to bound the problem.

3. Implementation

3.1 *Degeneracy*

In two dimensions a degeneracy will occur when four (or more) points in the structure are cyclic and thus their four territories meet at a point. In higher numbers of dimensions more subtle degeneracies can occur; for example, in three dimensions, when four points share a *line* in the Dirichlet tessellation.

Degeneracy is a problem, not so much when it actually occurs in the data, but when rounding error in the computer causes an algorithm to fail on a near degeneracy by, for example, making point P_i contiguous to point P_j but not making P_j contiguous to P_i. At what stages in the algorithm outlined above will degeneracies and near degeneracies be a problem?

As long as the new point to be inserted lies within the current convex hull there will be no difficulty about finding a deleted vertex at step one. The only problem will occur when the new point coincides with an existing point—this is easily avoided by requiring that each new point be at least some small distance from its nearest neighbour before it is considered for admission in steps 2-6. It will be remembered that the initial definition of the tessellation required that the points be distinct.

The tree search in step 2 is the only remaining stage in the algorithm where any floating point calculations are done. Once a list of deleted vertices has been found the remaining operations to be performed are all of a logical nature. Each candidate for inclusion in the list of deleted vertices is examined only once. The squared radius of the hypersphere of which a vertex candidate is the centre is compared with its squared distance from the new point (there is no reason to compare actual Euclidean distance—the square-rooting would waste time). If the new point is nearer to the vertex than its forming points are the vertex is added to the deleted vertex list. If it is further away the vertex is ignored. The only problem might arise when the squared radius and squared distance are equal (or nearly equal). Here a choice exists as to whether to include the vertex in the list or not. If the candidate vertex is included, the point opposite the line in the tessellation along which the tree search came to find the vertex will be considered contiguous to the new point but the section of territorial boundary between it and the new point will have zero area. That is to say the face of the new point's territory corresponding to its contiguity with the opposite point will be a polygon with k vertices all in the same place. Note that

1. The logic of the structure will be preserved, so computation may continue.
2. There is no numerical difficulty in calculating the position of these coincident vertices.

This argument can be extended to cover a multiple degeneracy. The program logic can keep the data structure as a valid tessellation and there will never be any possibility of, for example, division by zero when it is required to find the position of a degenerate vertex. A similar argument applies if the candidate vertex were rejected from the list of deleted vertices, except that here the result will be a line in the tessellation of zero length (in fact this is the course of action adopted in the author's implementation of his algorithm) as opposed to a face of zero area.

The second type of degeneracy occurs when $k + 1$ (or more) points lie in a hyperplane and are cyclic. This is a highly unlikely event unless the data points are intentionally placed on a regular grid (in which case it is usually easy to write the structure explicitly). Such configurations will cause problems if they occur as it is possible that, on the attempt at inserting the last of the $k + 1$ points, the vertex at one end of the line in the tessellation defined by the previous k points may be included in the deleted vertex list but the vertex at the other end may not. The program is then presented with the problem of finding a new vertex somewhere along the line, the position of which is, in effect, the point where the line crosses itself. Clearly the program might fail at this point. Fortunately this problem can be overcome as well. All that is necessary is to flag when it occurs (i.e. to flag the fact that the routines have been unable to compute the position of such a vertex) and always to include the vertex in the list of those deleted. This reduces the problem to a degeneracy of the first kind.

The author's algorithm has run successfully on highly degenerate point patterns (the points on a three dimensional sphere that can be obtained using three, four, five triangles, for example).

3.2 *Programming*

The algorithm has been implemented as a set of ISO FORTRAN subroutines that are callable from a simple main program that feeds them points one by one. The entire data structure of vertex lists and Delaunay simplexes is available to the user at any stage in this process. Various utility routines have also been written to do such things as producing lists of the vertices around a point's territory or common to a pair of points (the vertices associated with a contiguity). The package consists of about 900 lines of code, about a third of which are comments.

It is possible to make the usual compromises between storage space and execution time. For example it is possible to store the position and squared radius associated with each vertex in the structure, or to compute these values when they are needed.

In the author's implementation of his algorithm the floating point calculations have all been confined to two subroutines; one for calculating the squared distance between any two points in the k dimensional space in which the tessellation is being constructed, and the other for calculating the circumcentre and squared radius associated with the hypersphere that passes through the $k + 1$ points at the corners of a simplex.

This means that it should be possible to apply the algorithm to other than Euclidean spaces merely by changing these routines, as long as some precautions are taken. For example, on the surface of a sphere three initial points give rise to two vertices, in the plane three points give only one.

3.3 *Pictures*

Two figures have already been given showing the two dimensional version of the structure. It is a simple matter to take the three dimensional structure when it has been computed and process the co-ordinates of data points and tessellation vertices to produce a perspective image of the structure from some viewpoint. If the viewpoint is then moved slightly and another image produced the result will be a stereoscopic pair. **Fig. 4** shows such a stereoscopic pair of the tessellation of twenty points in a cube. Nineteen points were realised from the uniform distribution over the cube and then the twentieth was added at the cube's centre. The twentieth point and its associated territory were plotted using thicker lines than the rest of the structure to make it easier to see one complete territory. **Fig. 5** shows the corresponding Delaunay tetra-

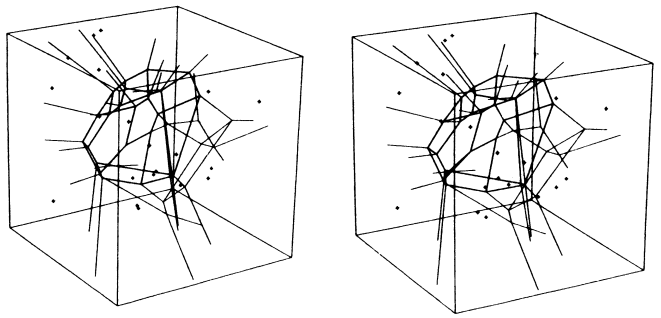

Fig. 4 The Dirichlet tessellation in 3 dimensions

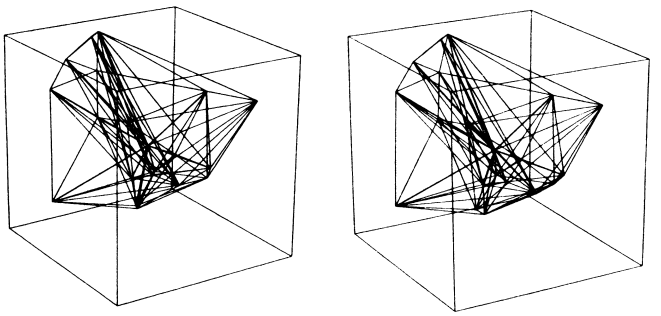

Fig. 5 The Delaunay triangulation in 3 dimensions

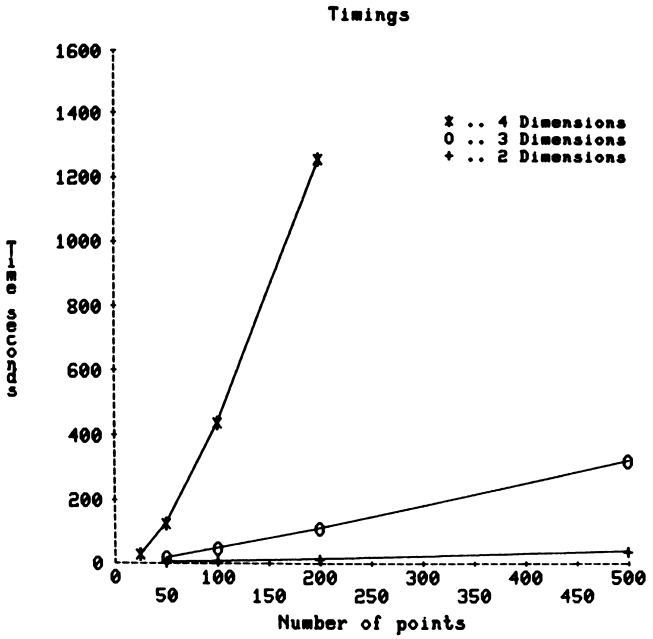

Fig. 6

hedrons. Again all the contiguities of the central point have been plotted using thick lines. If the reader does not have access to a stereoscope the three dimensional images can usually be seen by placing a piece of card between the left and right eye images on the page, relaxing the eyes so that the two images coalesce, and then attempting to focus the resulting single image. It is the second part of this process that is the more difficult. Very early in life the brain learns to link the angle between the eyes and the amount of distortion of the eye lenses needed to produce a sensible image. One solution is to use two identical magnifying glasses, one in front of each eye. This allows the eyes both to focus at and to point at infinity, whilst actually observing the image from a short distance.

The original plots of Figs. 4 and 5 used colour to differentiate the various aspects of the structures. Unfortunately printing difficulties make it impossible to reproduce them here.

3.4 *Timing*

As has already been mentioned, the work done in finding an initial vertex to delete is $O(n^{1/k})$. For n points in k dimensions this will lead to an $n^{(1+1/k)}$ term in the time taken to compute the tessellation. Once such a vertex has been found the time of computation per point is constant (given k) leading to an $O(n)$ term in the computation time. Thus the algorithm should take $O(a_k n^{(1+1/k)} + b_k.n)$ time to compute the structure for n points. The constants a_k and b_k will depend upon the initial choice of k. Simulations done by the author show that b_k in particular will increase quite rapidly with increasing k, as would be expected.

If the points were sorted before being presented to the algorithm (thus reducing the need for long nearest neighbour walks) the algorithm should take $O(a_k n.\log n + b_k n)$ time, though, in practice, the saving thus introduced is usually negligible.

Fig. 6 shows some timings of the algorithm for various numbers of points in 2, 3 and 4 dimensions. The time for the Green-Sibson algorithm operating on the same two dimensional 500 point data was 5·5 seconds as opposed to 28 seconds for the author's algorithm. As the Green-Sibson algorithm takes advantage of the two dimensional nature of the structure it would be expected to be faster, and the author would have no hesitation in recommending its use rather than his own algorithm for the specifically two dimensional case. All the runs were done on a twin processor Honeywell level 68DPS, owned by Bath and Bristol Universities, running the Multics operating system.

Acknowledgements

The author would like to thank the Social Science Research Council for the provision of the grant and research facilities that enabled him to carry out this work as part of their Spatial Data Research Project at Bath University. He would also like to thank Professor Robin Sibson and Dr Peter Green, who first introduced him to the Dirichlet tessellation and whose comments and suggestions have been invaluable.

References

BROWN, K. Q. (1979). Voronoi diagrams from convex hulls, *Inf. Processing letters*, Vol. 9 No. 5, p. 223.

COHEN, J. and HICKEY, T. (1979). Two algorithms for determining volumes of convex polyhedra, *JACM*, Vol. 26 No. 3, p. 401.

GILBERT, E. N. (1962). Random subdivisions of space into crystals, *Ann. Math. Stat.*, Vol. 33, p. 958.

GREEN, P. J. and SIBSON, R. (1978). Computing Dirichlet tessellations in the plane, *The Computer Journal*, Vol. 21, p. 168.

GREEN, P. J. and SILVERMAN, B. W. (1979). Constructing the convex hull of a set of points in the plane, *The Computer Journal*, Vol. 22 No. 3, p. 262.

LAWSON, C. L. (1972). Generation of a triangular grid with application to contour plotting, Cal. Tech. JPL Technical memorandum No. 299.

MILES, R. E. (1970). On the homogeneous planar Poisson process, *Mathematical Biosciences*, Vol. 6, p. 85.

RIPLEY, B. D. (1977). Modelling spatial patterns, *JRSS* Ser. B, Vol. 39, p. 172; Discussion, p. 192.

SIBSON, R. (1978). Locally equiangular triangulations, *The Computer Journal*, Vol. 21 No. 3, p. 243.

SIBSON, R. (1980). A vector identity for the Dirichlet tessellation, *Math. Proc. Camb. Phil. Soc.*, Vol. 87, p. 151.