Computing Dirichlet tessellations in the plane

P. J. Green and R. Sibson

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

A finite set of distinct points divides the plane into polygonal regions, each region containing one of the points and comprising that part of the plane nearer to its defining point than to any other. The resultant planar subdivision is called the *Dirichlet tessellation*; it is one of the most useful constructs associated with such a point configuration. The regions, which we call *tiles*, are also known as *Voronoi* or *Thiessen polygons*. We describe a recursive algorithm for computing the tessellation in a highly efficient way, and discuss the problems which arise in its implementation. Samples of graphical output demonstrate the application of the program on a modest scale; its efficiency allows its application to large sets of data, and detailed discussion of space and time considerations is given, based in part on theoretical predictions and in part on test runs on up to 10,000 points.

(Received September 1976, revised March 1977)

1. Introduction

1.1 *Definitions*

The Dirichlet tessellation is defined most simply as a subdivision of the plane determined by a finite set of distinct points: each point has associated with it the region of the plane nearer to that point than to any other. One might think of the points as being the locations of the lairs of competitive predators of equal strength; the region associated with each point is then the area available to the corresponding predator. Many analogous illustrations may be given; whether the model is a good one to describe spatial competition is an empirical question, and we are using that example here simply to illustrate the mathematical construct.

This definition applies more generally (indeed, the natural framework for it is a general metric space), and the construct is a useful one from a mathematical point of view in any number of dimensions (see Rogers, 1964). The one dimensional case is a triviality: the region associated with each point extends along the line in each direction from the point halfway towards the next point—or off to infinity at the endpoints. The exercise of trying to draw the tessellation in two dimensions for a few irregularly spaced points quickly reveals that the computational problem has already become nontrivial in the plane, and the object of this paper is to demonstrate an effective method of doing this computation. In three dimensions the problem acquires difficulties further to those encountered in two dimensions: the artifices (see Section 2.1) which make the two dimensional calculation an efficient one no longer apply and, although there is no doubt that the higher dimensional cases could be handled by an algorithm broadly similar to the one we describe, the detailed working out of such a procedure would be a formidable task which we make no attempt to undertake here.

The mathematical properties of the Dirichlet tessellation are conveniently set out in Rogers (1964). We give a formal definition slightly different from his. Let $P_1, P_2, \ldots, P_N$ be finitely many points in the plane, no two of which coincide. The *tile* of P_n is the set T_n defined by

$$T_n = \{x : d(x, P_n) < d(x, P_m) \text{ for all } m \neq n\}$$

where d is euclidean distance.

Every point of the plane with a unique nearest neighbour among $P_1, \ldots, P_N$ will lie in the tile of that nearest neighbour. Some points, however, will have two or more nearest neighbours, and our definition excludes these from membership of any tile. Each tile T_n is the intersection of the open half-planes bounded by the perpendicular bisectors of the lines joining P_n with each of the other P_m. Thus the tiles are convex and polygonal and may possibly (at the periphery of the structure) extend to infinity. Of course not all bisectors play an effective role in delimiting the tile (only those associated with 'nearby' points, intuitively); those that are effective each provide a straight line segment which also forms part of the boundary of a neighbouring tile. These boundary segments are the only parts of the plane not within any tile, and comprise precisely those points with two or more nearest neighbours among $P_1, \ldots, P_N$.

Some authors use 'polygon' for 'tile' and the names Dirichlet, Voronoi, and Thiessen are variously associated with the construction, which we call the *Dirichlet tessellation*. Tiles which have a boundary segment in common are said to be *contiguous*, as are their generating points. In general tiles meet in threes at vertices so the lines joining contiguous generating points define triangles; these triangles can easily be shown to fit together into a triangulation of the convex hull of the generating points; the perpendicular bisectors of the edges of this triangulation give the boundaries of the tiles, and the circumcentres of the triangles are the vertices of the tiles. The triangulation is called the *Delaunay triangulation*. *Mutatis mutandis*, all of the above extends to higher dimensional cases. **Fig. 1** shows the tessellation and triangulation associated with a small scale set of points in the plane.

1.2 *Windows*

As we have defined it, the Dirichlet tessellation lives in the whole plane and in fact our definition extends from a finite to a locally finite set of generating points. In practice, however, it is usual for the data to be collected or constructed and the tessellation computed within a restricted 'window'.

If the window excludes part of the plane, but none of the points, then its effect is literally that of blanking out what goes on outside it: each windowed tile is the intersection of the window with the corresponding unwindowed tile. This can affect the contiguities, for the common boundary segment of tiles of points near the periphery can begin a long way outside the convex hull of the set of points. This phenomenon should not be regarded as any kind of defect; indeed, part of the merit of working within a window is to prevent the generation of contiguities which would surely have been eliminated if observations had been taken over a larger region.

A window which excludes some of the points can have a more drastic effect; were such a point to be included in the construction its tile could well lie partly within the window, although it itself lay outside. The behaviour of this effect cannot be concisely described; we shall not consider it further in this computational context, but will rather make the convention that points discovered to lie outside the window will be *rejected* altogether, the computation proceeding as if they did

not exist. The remaining points are described as *accepted.* **Fig. 2** illustrates the effects of windowing.

Our formal definition of the tessellation corresponding to points $P_1, \ldots, P_N$ and window E is that the tile T_n^* of the point P_n is given by

$$T_n^* = \{x \in E : d(x, P_n) < d(x, P_m) \text{ for all } m \neq n, P_m \in E\}$$

if $P_n \in E$, and is otherwise undefined. If difficulties over disconnected tiles are to be avoided, it is desirable to restrict attention to convex windows and to ensure easy specification and manipulation to make them polygonal. In fact many applications call for a square or rectangular window. For technical reasons we require the window to be a nonempty open polygonal region defined by some number of strict linear inequality constraints $ax + by + c < 0$. Any finite number of such constraints may be specified: each is described as *effective* if its omission would change the content of the window, otherwise as *redundant.* Redundant constraints may be completely ignored. It is positively convenient to operate within a window that is bounded, and we do in fact insist on this.

The observation made above about the effect of a window that contains all of the points shows how to deal with an arbitrary (non-convex or non-polygonal) bounded window: simply operate within a convex polygonal window containing it and impose it afterwards. Similarly, unbounded windows can be simulated by sufficiently large bounded ones.

1.3 *Discussion*

Like so many other simple mathematical ideas, the Dirichlet tessellation has been reinvented on a number of occasions, sometimes in the context of a particular type of application,

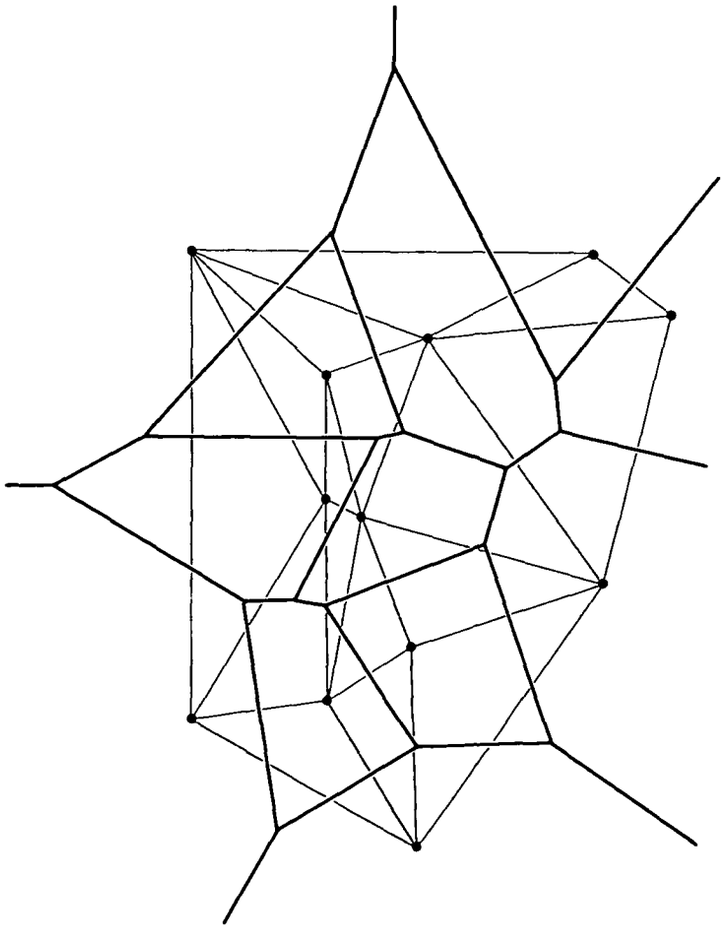

Fig. 1 The Dirichlet tessellation (bold lines) and Delaunay triangulation (fine lines) for a small scale configuration

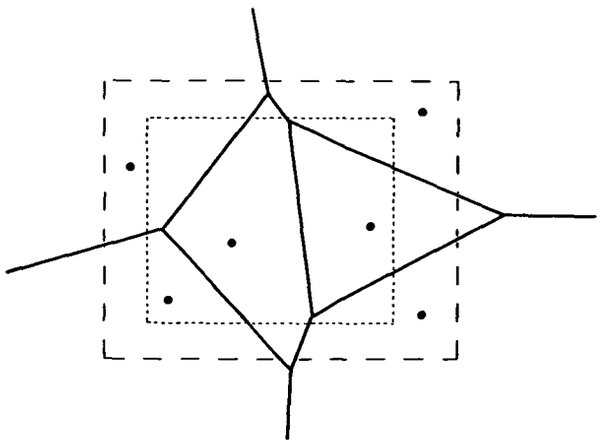

Fig. 2 The effects of windowing: the broken-line window excludes some contiguities but no points; the dotted-line window excludes some points but not the whole of their tiles

and this accounts for the diversity of names under which it may be encountered. Rogers (1964) discusses the mathematical background. Geographers speak of Thiessen polygons; applications are described by Rhynsburger (1973). Crystallographers recognise a number of models for crystal growth: one of these, which is associated with the name of Meijering, gives rise to the Dirichlet tessellation; a convenient reference is Gilbert (1962). The construct is now becoming quite well known to statisticians: for example, applications in plant ecology are discussed by Mead (1971); Besag (1974) points the way towards using the contiguities as a replacement for the self-evident 'neighbour' relationship of points on a regular lattice in the construction of probabilistic models for spatial phenomena; and this has also been explored by Ripley (1977). Miles (1970) derives various statistical and ergodic properties of the Dirichlet tessellation and Delaunay triangulation constructed from a homogeneous Poisson process in the whole plane. The production of contour maps based on observations over an irregular grid seems likely to become an important application (McLain, 1976; Powell and Sabin, 1977). In this context, Lawson (1972) has suggested the use of triangulations with the following local equiangularity property: in every convex quadrilateral formed by two adjacent triangles, the minimum of the six angles in the two triangles is not less than it would have been had the alternative diagonal and pair of triangles been chosen. Sibson (1978) has shown that the Delaunay triangulation uniquely possesses this property. We ourselves intend to explore applications in the fitting of smooth surfaces (*cf.* Wahba and Wold, 1975), scaling (*cf.* Kendall, 1971) and spatial cluster analysis. We feel that it is no exaggeration to claim that the Dirichlet tessellation is one of the most fundamental and useful constructs determined by an irregular lattice.

In this paper, we give no further discussion of its application; we confine ourselves to the problem of its computation. An illuminating, if slightly distressing, account of the history if this problem is given by Rhynsburger (1973), who writes of methods based on subjective judgments or arbitrary (and in fact incorrect) rules and records the disappointment with which their inventors encountered re-entrant polygons in the course of their work! He concludes his paper by proposing an algorithm which is correct, but admittedly provisional and inefficient. Several other papers are of computational interest—Mead (1971), Lawson (1972) and McLain (1976) are worth noting—but in no case is an efficient general purpose algorithm offered, and we have not found in the literature anything which claims to be a definitive treatment of the

problem. A general discussion of geometrical algorithms is given by Shamos (1975); among the many problems which he considers is that of constructing the Dirichlet tessellation. He outlines an algorithm for this which might in principle prove reasonably efficient, but this is difficult to assess in the absence of any actual implementation, and his proposal is not given in sufficient detail to make all the steps of an implementation plain. Finally, we are aware of the existence of one unpublished and uncirculated program which we believe to be of reasonable efficiency, using some similar ideas to our own, and there may be others, but we have had no access to any such material.

2. The Algorithm

2.1 *The recursive step*

The algorithm that we implement is a recursive method for computing contiguities. All other properties of the tessellation, tile vertex positions and tile areas for example, are easily calculated from the contiguities and the point coordinates. Contiguities between tiles were discussed in the introduction; recording the existence of a contiguity may be regarded as a way of labelling an edge of a tile. Each edge is part of the boundary of two tiles and access to it from either side is needed, so for each member of a pair of contiguous points the contiguity with the other has to be recorded. For points on the periphery an extra complication arises: the tile of such a point is bounded in part by segments of effective constraints rather than by inter-tile edges. In fact this is simply dealt with, by making no distinction between constraints and points as far as contiguity is concerned. They are considered together as *objects*, and for each object is recorded a *contiguity list*, that is a list of the objects to which it is contiguous.

This has the slightly unexpected consequence that each effective constraint must itself have a contiguity list, some of the objects in which are points and others—exactly two—effective constraints. The parallels between points and constraints actually carry further than this: when a contiguity between a point and a constraint is being considered, that constraint may temporarily be replaced by a *virtual point*, namely the reflection in it of the true point; the constraint is then the perpendicular bisector of the line joining the true and virtual points. It turns out that this completely resolves the problem of manipulating constraints, since it is never necessary to replace a constraint by more than one virtual point at a time. The obvious need to avoid duplicated points carries over to the requirement that a point and its reflection should never coincide; that is why the window must be open.

All of these remarks apply equally to any number of dimensions. What makes the two dimensional case special is that the contiguities associated with an object can be recorded in cyclic order (by convention, anticlockwise). For points this cyclic order has no natural starting point; we have a ring rather than a list and we break this ring arbitrarily. For constraints the two adjacent contiguities with other constraints mark a natural break-point, but it is unnecessary and indeed inconvenient to maintain this. Cyclic order is more than an administrative convenience; it is the basis on which the possibility of economical computation rests.

The method is to scan the points in turn, recursively modifying the contiguities as each point is added. The recursive step is straightforward to describe in geometrical terms, although some care has to be taken over actually implementing it computationally. The information for recursion consists of the details of the window, the coordinates of the previously accepted points and the contiguities that these objects determine. The coordinates of the new point, which are saved after use, are used to update the contiguities on the inclusion of the new point. It is easy to visualise the effect of the new point: it acquires its tile by winning territory from the tiles of nearby

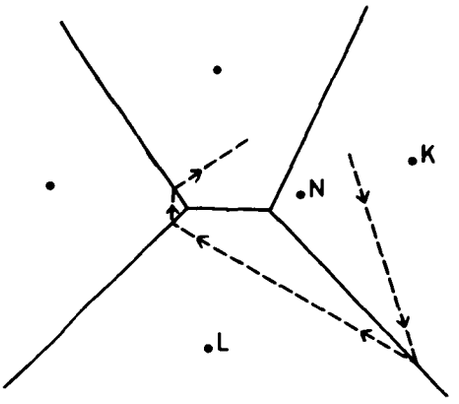

Fig. 3 Growing the tile round a new point N; its nearest neighbour is K, and the first point to go on its contiguity list is L

points, in fact of precisely those points with which it subsequently proves to be contiguous. It is these points and these only whose contiguities are affected by the introduction of the new point.

One point which is certain to be contiguous to the new point is its nearest neighbour among the old points; this point (or any such nearest neighbour if there is more than one) must first be found. The best way to do this is a problem we discuss in a subsequent section; for the moment, suppose it done.

The contiguity list for the new point is then built up in reverse (that is, clockwise) order and subsequently standardised. We begin by finding where the perpendicular bisector of the line joining the new point to its nearest neighbour meets the edge of the nearest neighbour's tile, clockwise round the new point. Identifying the edge where this happens gives the next object contiguous with the new point and this is in fact the first to go onto its contiguity list. The new perpendicular bisector is then constructed and its incidence on the edge of this new tile is examined to obtain the subsequent contiguous object; successive objects are added to the contiguity list in this way until the list is completed by the addition of the nearest neighbour. Whilst this is being done old contiguity lists are being modified: the new point is inserted in each and any contiguities strictly between the entry and exit points of the perpendicular bisector are deleted, the anticlockwise-cyclic arrangement of the lists making both this and the determinaton of the exit very easy. **Fig. 3** illustrates the start of this process.

An appealing feature of the recursive nature of our algorithm is that the coordinates of a point are not needed until that point is considered for inclusion. Even the number of points may be *a priori* unknown, if storage can be arranged dynamically; and the algorithm could be extended to remove points as well as add them. This suggests various applications, among them real time analysis of spatial data and adaptive quadrature on an irregular grid.

2.2 *Degeneracies*

It may happen that four or more tiles meet at a vertex; such a vertex is said to be *degenerate*. A regular square lattice determines a tessellation in which every vertex is degenerate; but there is little need to use a tessellation algorithm in such a case and it is unlikely that degeneracies would occur in the kinds of data to which the algorithm would be applied. What may occur more commonly are near-degeneracies and the importance of handling degeneracies correctly stems more from the need to deal consistently with such cases than from any expectation that actual degeneracies will occur.

The defining points of the four or more tiles meeting at a

degenerate vertex must all lie on a circle with that vertex as centre. If we consider a vertex of degree four, it can be seen that degeneracy is a transitional state: the surrounding contiguities form a quadrilateral and as this is deformed through cyclic shape, the one diagonal disappears and is replaced by the other. We believe that it is of little importance if for numerical reasons the diagonal contiguity is recorded incorrectly when the situation is close to degenerate. What is disastrous is to record it inconsistently. An inconsistency could take the form either of a contiguity being recorded from one side but not the other; or of both diagonals of the quadrilateral being recorded as contiguities. The resolution of these problems proves to be a natural consequence of efficient programming in that the way to avoid difficulties is to ensure that each calculation step is carried out once only with any necessary information for later use being saved from it; disaster can strike only if two algebraically equivalent but numerically distinct calculations are made. We believe that prudent design of the implementation has made it almost impossible that numerical errors could lead to inconsistencies, and we have encountered no difficulties when testing our program on degenerate data.

Should a vertex turn out to be exactly numerically degenerate neither diagonal is recorded as a contiguity; recall that tiles are contiguous only if they meet along a boundary segment and not just at a vertex.

2.3 *Finding the nearest neighbour*

We have observed that in the majority of applications, every vertex has degree three. If this holds, use of the Euler-Poincaré formula *faces* − *edges* + *vertices* = 2 (for the sphere; for the plane the infinite region is counted as a face) gives an exact expression for the total number of contiguities which have to be recorded, namely

$$4 \times (\textit{effective constraints}) + 6 \times (\textit{accepted points}) - 6 ,$$

counting each twice, once from each side. Thus not only does the average number of contiguities per point approach six in large configurations, irrespective of the positions of the points provided only that there are no degeneracies, but also the number of contiguities added by the introduction of a new point is exactly six. Degeneracies serve only to reduce the total number of contiguities. Therefore the workload of growing the tile round a new point once the nearest neighbour has been found is on average constant—it does not grow with the number of points previously accepted. We have verified this prediction empirically. This component of the work performed by the algorithm thus gives rise in the total workload to a term linear in the number N of points.

It is therefore particularly important to have an efficient method of finding the nearest neighbour; a naive search would lead to an $O(N)$ term per point, and thus to an $O(N^2)$ term in the total workload and that is not good enough. The technique which is obvious as an improvement on this is to use the tessellation constructed so far as a guide to the relative positions of the points. Simply start at an arbitrary point and 'walk' from neighbour to neighbour, always approaching the new point, until the point nearest to it is found. Implementing this process is easy; its effectiveness depends on how good a guess the starting point of the walk is. In some applications the list of points is automatically in a fairly systematic order because of the way the data have been collected. Under these circumstances the previous point is likely to be close to the new one and the use of this as the starting point results in an almost negligible workload. If nothing systematic is known about the positions of the points, it is sensible to start from a reasonably central point and one would expect that for each point this would result in an $O(N^{\frac{1}{4}})$ term, total $O(N^{\frac{5}{4}})$. This appears to be the case in practice. It is the technique used in our implementation and although the resultant term is ultimately the dominant one in the workload it is not until over 7000 points have been added that the work of finding the nearest neighbour becomes equal to the rest of that of adding a point.

By walking across several generations of tessellations—saving the tessellation every time the number of points increases by some suitable factor—one would expect to be able to reduce $O(N^{\frac{3}{2}})$ to $O(N \log N)$, at a cost in storage. Although attractive in theory this appears to be pointless in practice for problems of the scale we have considered so far, but if a need for it is established it would not be difficult to implement. Readers familiar with sorting algorithms will no doubt find the comparison with tessellation interesting; there are both helpful analogies and significant differences. In both cases it is possible to think in terms of locating and then inserting a point. The workload for insertion is constant, but location becomes more and more difficult as N increases—$O(N)$ per point for the bubble sort, $O(N^{\frac{1}{2}})$ per point for the planar nearest neighbour walk. In both cases this workload can be reduced by taking long steps towards the right position early in the process. Shell's sorting method does this for all points at once without using extra storage but we have not been able to devise any analogue of this for tessellation. The multiple generation scheme can be implemented both for sorting and for tessellation; there is a cost in storage, but the recursive property is retained, and the workload per point may in both cases be expected to be reduced to $O(\log N)$. Of course, the tessellation problem has geometrical aspects of far greater complexity than sorting.

3. Implementation

3.1 *Preliminary considerations*

The fundamental problem encountered in implementing the algorithm described above arises from the need to store and update lists of contiguities. No contiguity list is of length less than three. In principle the only upper limit is the sum of the number of points and the number of constraints, or at least something close to that. In practice a point seldom has more than ten contiguities; in typical data we have found this to occur about once in six hundred points. Both the content and length of a list may change on updating.

If the algorithm were being implemented to solve one specific problem, then clearly it would be appropriate to minimise programming effort by choosing a language which would handle automatically all the necessary housekeeping. However when it is desired to produce a highly efficient, compact, portable and compatible implementation for regular general use, such languages seem less attractive. We have not found it unduly burdensome to work in ANSI FORTRAN in setting up an implementation and this has allowed us to take full advantage of the special structure of the problem so as to reduce as far as possible the space occupied by administrative information and the time needed to maintain it.

Contiguity lists are kept in a single integer vector, the *heap*, with address pointers to and from addressing arrays for points and for constraints. Rather than modify existing contiguity lists as points are added new versions of these lists, and all new lists, are created at the base of the free space in the heap and old versions are flagged as dead. When the heap becomes full dead space is returned to the top end as free space by a straightforward garbage collection routine.

Although these are not FORTRAN-like concepts, they are easy to set up within FORTRAN; they give rise, for example, to none of the difficulties which are encountered in any attempt to handle characters or individual bits.

3.2 *Program organisation*

Our implementation is in the form of a self-contained family of ANSI FORTRAN subroutines and functions with no I/O and extends to about 700 lines of which one-third are comments.

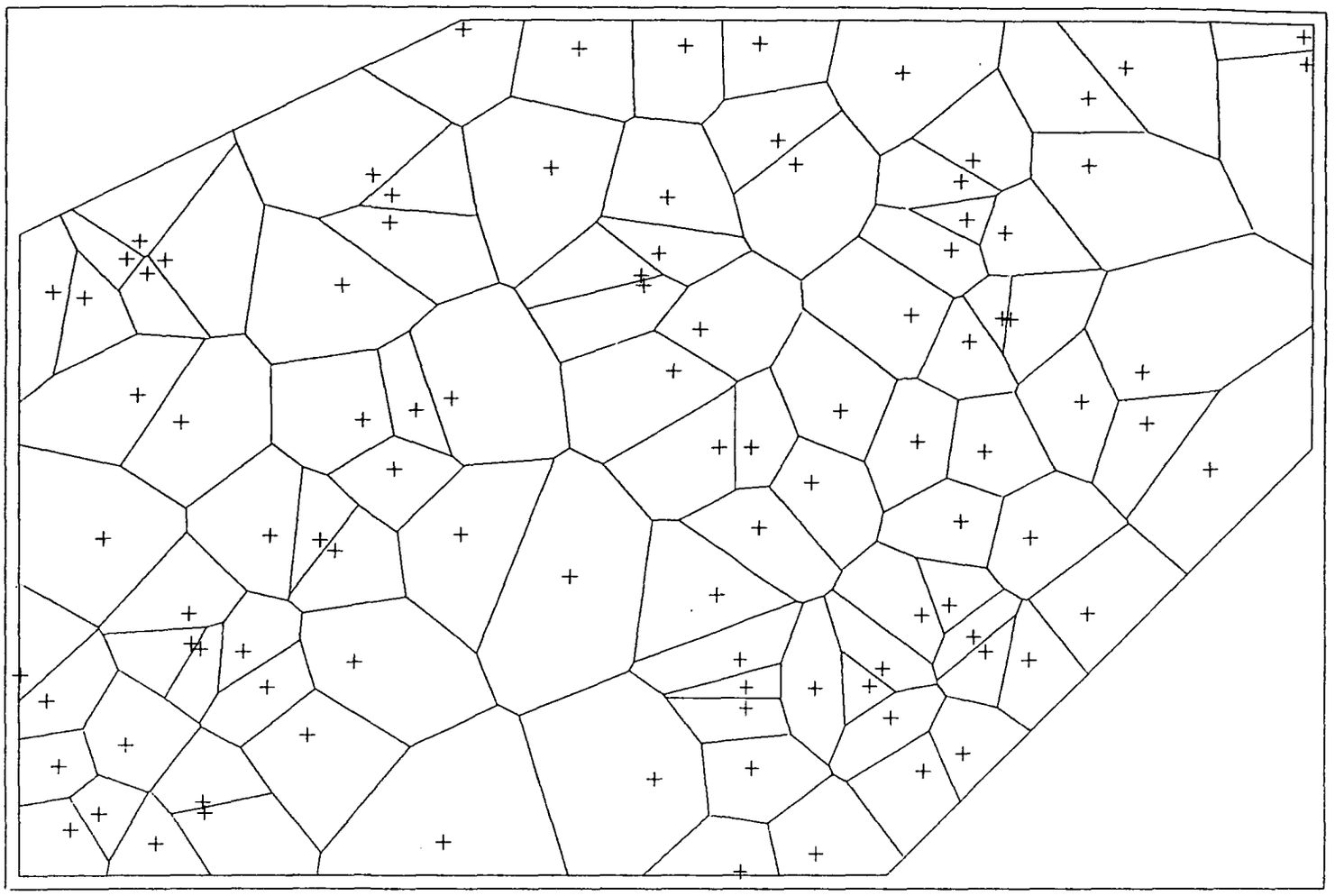

Fig. 4 The Dirichlet tessellation for 100 points within a window. The points are realised independently from the uniform distribution over the window

The user calls a single master subroutine TILE. This calls internal subroutines which first construct a sensible description of the window, then set up the data structure on the detection of the first acceptable point and then add points recursively until the tessellation is complete. Garbage collection takes place as required and again before control is returned to the calling program. The starting point for the nearest neighbour walk is occasionally updated to lie near the centroid of the accepted points; this helps to minimise the adverse effect of any partial systematic order in the labelling of the points.

No substantial numerical difficulties are encountered in the implementation and single-precision (even 24-bit mantissa) arithmetic has proved quite acceptable. Care is needed in arranging the calculation of intercepts of perpendicular bisectors on one another so that difficulties over degeneracy are avoided, but we have not succeeded in provoking any unacceptable responses from these numerical routines.

Errors in the data may cause various conditions invalidating the construction of the tessellation. These are detected and identified, as also is the condition of heap overflow; they cause an error flag to be set and control is returned to the calling program.

3.3 *Storage and times*

The information which has to be stored comprises the point coordinates, the contiguities and the information for administering the contiguities. Normally there are only a few constraints and the space required to handle them may be regarded as part of the fixed overheads. The data dependent storage is as follows. For each point we require:

two real locations for its coordinates
one integer location to address its contiguity list in the heap
sufficient space in the heap.

The typical length of a contiguity list is six. Two more locations per point are required to administer the data, so the minimal possible heap size is about eight times the number of points. We find that a heap size of nine times the number of points gives adequate space for this information together with the relatively small amount of boundary information also kept there and also allows enough working space to ensure that the time taken by garbage collection is kept reasonably small. Thus twelve locations per point are needed, of which only four contain administrative information. On a byte-based machine the heap can consist of two-byte integers, giving a total storage requirement of thirty bytes per point.

General considerations governing the behaviour of the run times have been discussed earlier. It may be of interest to report some actual times. The program proves to be very fast. We have made many test runs on 1000 points. On the CDC7600 at the University of London Computer Centre, running under the optimising FORTRAN compiler, such runs take a little under 0·8 sec. with a heap size of 10,000. Runs on 9000 points with heap size 90,000 took 9·7 sec. and on 10,000 points with the same heap size, 11·6 sec. Storage rather than time is clearly the limiting factor. On other machines we have carried out runs on up to 12,000 points. Variations in compiler performance make comparative speed predictions difficult but the usual

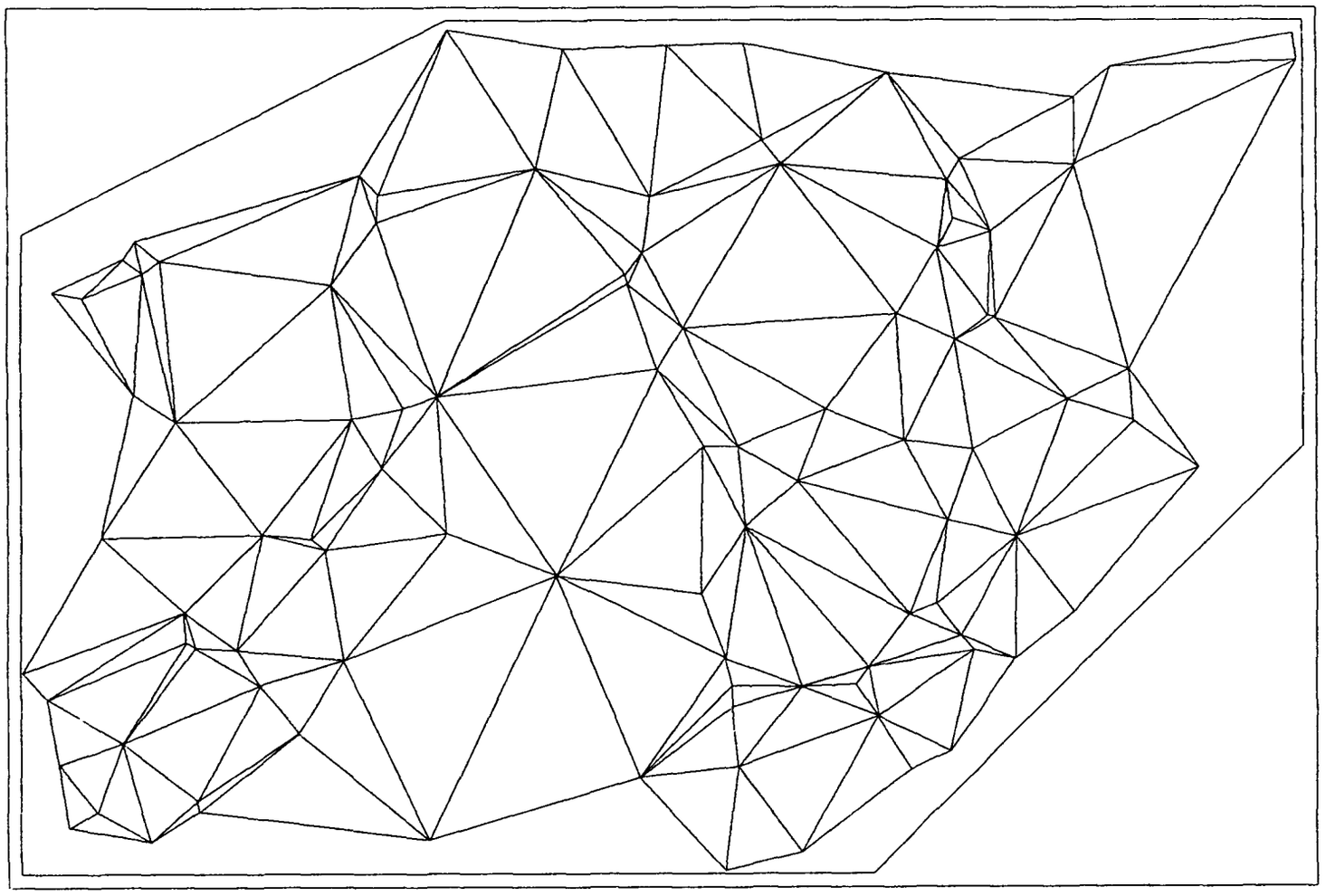

Fig. 5 The Delaunay triangulation for the same set of points as in Fig. 4

benchmark comparisons give a first approximation.

3.4 *Auxiliary plotting program*

The tessellation algorithm may be expected ultimately to enter into a variety of sophisticated procedures for data analysis, and the development of such procedures is a matter which will be explored by the authors in future papers. A general auxiliary program which is, however, worth mentioning at this stage is one to plot the tessellation and/or triangulation. We have prepared a program for this purpose which requires supplementation by local interface routines for particular installations. We certainly do not claim optimal efficiency for the plotter program, but it is at least not embarrassingly inefficient; the amount of pen-up movement is far less than that which would be generated by a program which did not take the geometry into account. **Figs. 4** and **5** illustrate its output.

3.5 *Availability*

Potential users of this program are invited to contact the authors.

References

BESAG, J. (1974). Spatial interaction and the statistical analysis of lattice systems, *Journal of the Royal Statistical Society*, Series B, No. 36, pp. 192-225, discussion pp. 225-236.

GILBERT, E. N. (1962). Random subdivisions of space into crystals, *Annals of Mathematical Statistics*, Vol. 33, pp. 958-972.

KENDALL, D. G. (1971). Construction of maps from 'odd bits of information', *Nature*, London, Vol. 231, pp. 158-159.

LAWSON, C. L. (1972). Generation of a triangular grid with application to contour plotting, California Institute of Technology Jet Propulsion Laboratory, *Technical Memorandum* #299.

MCLAIN, D. H. (1976). Two dimensional interpolation from random data, *The Computer Journal*, Vol. 19, pp. 178-181.

MEAD, R. (1971). Models for interplant competition in irregularly spaced populations, In *Statistical Ecology* (G. P. Patil *et al.*, eds), Volume 2, pp. 13-30, Penn State University Press.

MILES, R. E. (1970). On the homogeneous planar Poisson process, *Mathematical Biosciences*, Vol. 6, pp. 85-127.

POWELL, M. J. D. and SABIN, M. A. (1977). Piecewise quadratic approximations on triangles, *ACM Transactions on Mathematical Software*, Vol. 3, pp 316-325.

RHYNSBURGER, D. (1973). Analytic delineation of Thiessen polygons, *Geographical Analysis*, Vol. 5, pp. 133-144.

RIPLEY, B. D. (1977). Modelling spatial patterns, *Journal of the Royal Statistical Society*, Series B, Vol. 39, pp. 172-192, discussion pp. 192-212.

ROGERS, C. A. (1964). *Packing and Covering*, Cambridge Mathematical Tracts #54, Cambridge University Press.

SHAMOS, M. I. (1975). Geometric complexity, *Proceedings of the 7th SIGACT Conference*, Albuquerque, New Mexico, pp. 224-233.

SIBSON, R. (1978). Locally equiangular triangulations, *The Computer Journal*, to appear.

WAHBA, G. and WOLD, S. (1975). A completely automatic French curve: fitting spline functions by cross-validation, *Communications in Statistics*, Vol. 4, pp. 1-17.