LBL-32989
UC-405
Preprint

Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Physics Division

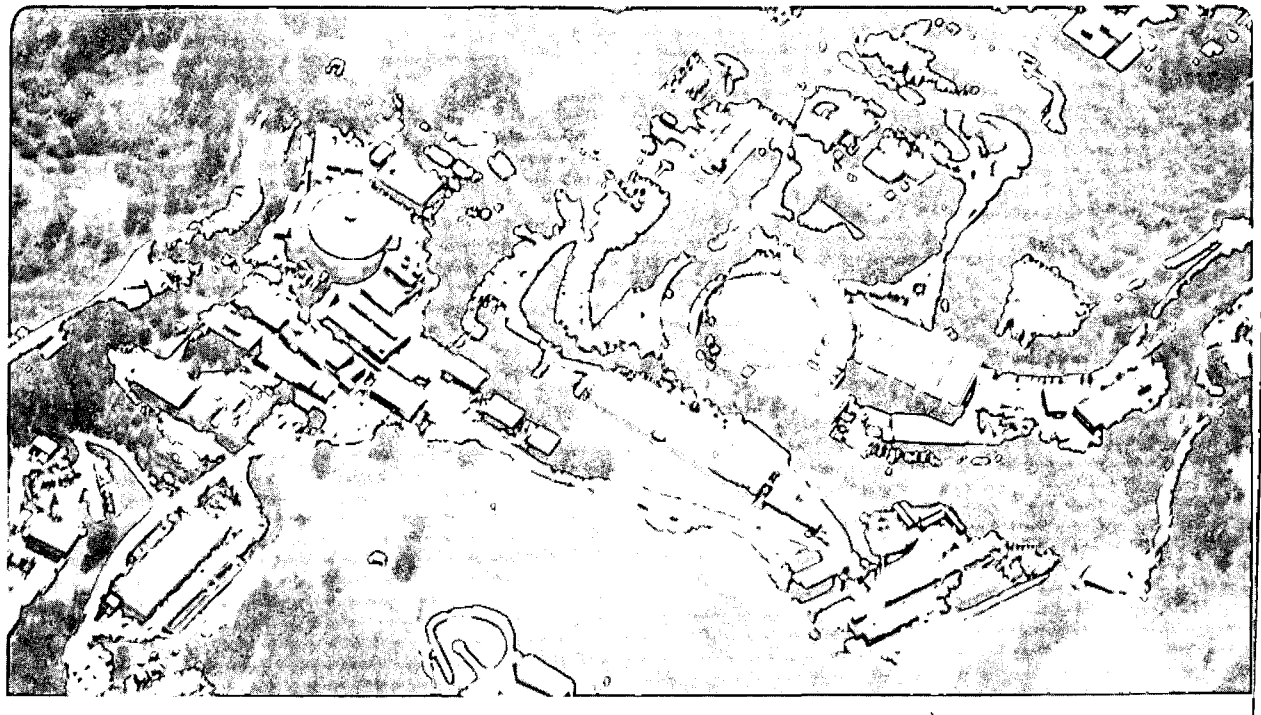
Mathematics Department

To be submitted for publication

An Adaptive Cell Method for Delaunay Triangulation

J. Strain

March 1991



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098

DISCLAIMER

LBL-32989

An Adaptive Cell Method for Delaunay Triangulation *

John Strain
Department of Mathematics
Princeton University
Princeton, NJ 08544

20 March 1991

AMS(MOS) Subject Classifications: 65N50, 68C05, 68C25, 65D05, 52A45, 05B45.

Keywords : Delaunay triangulation, Voronoi diagram, Dirichlet region, Thiessen triangulation, cell method, fast algorithms, adaptive methods.

Electronic Mail: strain@math.princeton.edu

* This work was supported in part by a NSF Mathematical Sciences Postdoctoral Research Fellowship and in part by the Applied Mathematical Sciences Subprogram of the Office of Energy Research, U.S. Department of Energy under Contract DE-AC03-76SF00098 while the author was visiting Lawrence Berkeley Laboratory.

Abstract

We present two new fast algorithms for constructing the Delaunay triangulation of a set of N points in the plane. The first algorithm is based on a uniform cell approach; it runs in $O(N \log N)$ expected time when the points are uniformly distributed and $O(N^2)$ otherwise. The second algorithm is an adaptive cell method which runs in $O(N \log N)$ time even when the points are drawn from a highly nonuniform distribution. Both algorithms extend immediately to three-dimensional Delaunay tessellation.

We also describe numerical experiments which show both methods to be highly efficient when N is larger than a hundred. The adaptive method is slightly slower than the uniform on uniformly distributed points but hundreds of times faster when N is large and the points are nonuniformly distributed. It is $O(N \log N)$ in either case, while the uniform method degenerates to $O(N^2)$ for nonuniformly distributed points.

1 Introduction

The Voronoi diagram and its dual, the Delaunay triangulation, are important tools in scientific computing. They are used in:

- computational chemistry [36]
- crystal growth modelling [2, 3, 34]
- free Lagrangian methods in computational fluid dynamics [6, 16, 15, 17, 26]
- interpolation of data given at irregularly arranged data points [1, 20, 24, 28, 29, 38]
- nearest-neighbor problems in computational geometry [31, 5, 27]
- particle methods for fluid mechanics and transport problems [30]
- triangulation for finite element methods [19, 39, 22, 33]

For applications to geography, ecology, statistics, pattern recognition, and other areas, see [4] and the references therein.

Many practical situations require construction of the Voronoi diagram or Delaunay triangulation for a set of N points in $\mathbf{R}^d$ where $d = 2$ or 3 and N can be as large as 10^6. This can be a very expensive calculation, requiring hours of CPU time and large amounts of memory. Thus it is important to have fast algorithms for constructing the Delaunay triangulation. Previous work in this direction is discussed in §2.2, after a brief survey of the relevant properties of the Delaunay triangulation in §2.1.

The main body of the paper begins in §3, where we present a uniform cell method for constructing the Delaunay triangulation in $\mathbf{R}^2$. This method is quite efficient when the points are more or less uniformly distributed, and is fairly straightforward to program. It also extends immediately to the three-dimensional case, unlike many of the fast algorithms discussed in §2.2. The algorithm also extends to solve constrained Delaunay triangulation and other local geometric problems.

In §4, we present an *adaptive* cell method which triangulates non-uniformly distributed points efficiently. Unlike the uniform cell method, it is not slowed

down by normally distributed points. Like the uniform cell method, it extends to three dimensions and to other local geometric problems.

The results of numerical experiments with both algorithms are presented in §5. They confirm the $O(N \log N)$ timing of the uniform method on uniform points and the $O(N \log N)$ timing of the adaptive method on non-uniformly distributed points.

In §6, we discuss refinements and generalizations of the methods presented in §3 and §4. These include different search strategies and other geometric problems. Finally, in §7, we discuss our conclusions.

2 Delaunay triangulation

For the history and development of Voronoi diagrams and the Delaunay triangulation, the reader is referred to [4]. Here, we confine ourselves to describing a few useful properties in §2.1 and describing some of the many previous algorithms for construction of the Voronoi diagram or Delaunay triangulation in §2.2.

2.1 The Delaunay triangulation

Suppose $X = \{x_j : j = 1, 2, \ldots, N\}$ is a set of N points in a set $\Omega \subset \mathbf{R}^2$; for convenience we assume Ω has a polygonal boundary. The *Voronoi diagram* $V(X)$ of X is the set of polygons V_j defined by

$$V_j = \{x \in \Omega : |x - x_j| \leq |x - x_i| \text{ for all } i \neq j\}. \tag{1}$$

Thus V_j is the set of points in Ω which are closer to x_j than to any other point x_i in X. The Voronoi diagram of X is a useful tool for identifying nearest neighbors, because the nearest neighbors of x_j are precisely those points x_i whose Voronoi polygons V_i share an edge with V_j. The Voronoi diagram is used to solve closest point problems in computational geometry, for precisely this reason, in [5].

The dual of the Voronoi diagram is the Delaunay triangulation, obtained by connecting two points with a triangle edge iff their Voronoi polygons share an edge. (This prescription breaks down if four or more points of X lie on a circle; then some edges of their Voronoi polygons have zero length. Then one can triangulate the cocircular points in any nondegenerate way, so the resulting Delaunay triangulation is not unique. This possibility requires careful treatment in numerical calculations, because the topology of the Delaunay triangulation can change by passing through such a case [36].)

The Delaunay triangulation is distinguished among all possible triangulations of X by certain properties; for example, it is locally equiangular [32], also, the circumcircle of any of its triangles contains no other point of X in its interior [20]. It has been criticized as a tool for finite element calculations because it contains too many long thin triangles near boundaries [11, 33], but it is proved almost optimal for interpolation of scattered data in [38]; the Delaunay triangulation nearly maximizes the minimum angle and minimizes the error bounds for linear interpolation over all triangulations of

X, as a consequence of the minimum-angle property proved in [20]. This controversy seems to spring from the dichotomy between constructing a nice triangulation of a set of given points and constructing, as one does in finite element calculations, the points and the triangulation simultaneously. When the points are fixed, the Delaunay triangulation is one of the best possible triangulations; but when points can be added or deleted at will, better finite element meshes can be constructed.

Many generalizations of the Delaunay triangulation have been introduced. For example, there is much interest in the constrained Delaunay triangulation [10], in which some edges and some barriers (which edges may not cross) are given as well as the points X to be triangulated. It is useful in motion planning, for obvious reasons, and can be constructed with the methods of this paper.

2.2 Previous Algorithms

Many authors have constructed fast algorithms for the Delaunay triangulation or the Voronoi diagram. (With enough information about either, it is straightforward to construct the other.) These algorithms are based on various principles and transformations; they can be roughly classified as follows:

- Diagonal swapping methods [16, 17] usually start with some reasonable triangulation of X and swap diagonals until the Delaunay triangulation is reached. That is, they inspect the quadrilateral formed by each pair of adjacent triangles to see if the triangulation can be improved by reconnecting its diagonal. Finite termination of this method was proved in [20]. The more sophisticated approaches in [20, 28, 9] determine the convex hull simultaneously to achieve $O(N^{4/3})$ running time. These methods have some advantage in time-dependent problems because X may change only slightly from one time step to the next, giving a natural starting point for swapping. But each quadrilateral still needs to be checked, so the advantage is not as great as one might hope.

- Divide-and-conquer methods which construct the Voronoi diagram or Delaunay triangulation in worst-case $O(N \log N)$ time are described in [31, 13, 21]. These methods split the set X into two or more subsets, form the Voronoi diagram or Delaunay triangulation of each subset separately, and merge the results to obtain the whole object. Applying this technique recursively produces an $O(N \log N)$ algorithm, which is

asymptotically optimal in a common model of computation. (Maus [23] points out that one can do better in finite precision.) The difficulty in applications seems to be the difficulty of programming the merge step efficiently. Thus these methods seem to be mainly of theoretical interest so far. See, however, [22] which combines a divide-and-conquer method for constructing the initial triangulation with diagonal swapping to make it Delaunay.

- Incremental methods which construct the Voronoi diagram by adding one point at a time have been very popular; see [7, 18, 36, 26, 37]. These methods have been extended to a periodic geometry and applied to fluid mechanics in [6]. This latter algorithm, like most incremental algorithms, is not worst-case optimal, but can update $V(X)$ in $O(N)$ time if X is not too different from a configuration for which the diagram is known and certain extra information is saved. Dually, the methods presented in [24, 23] find the Delaunay triangulation one triangle at a time, typically by a local optimization procedure as summarized in [20]. The method presented in the current paper is based on the local optimization technique introduced in [24], speeded up with a cell approach.

- Sweepline methods were introduced in [14]. They seem to combine some of the simplicity of incrememental methods with the $O(N \log N)$ worst-case behavior of divide-and-conquer, and thus seem likely to be quite useful in practice.

- Cell methods for Voronoi diagrams have been used in [5, 25]. These methods combine incremental construction of the diagram with a data structure which organizes the points of X into "cells" by spatial location. They run in $O(N \log N)$ *expected* time when the points of X are drawn from a quasi-uniform distribution on a bounded set, because construction of the Voronoi diagram is then a local problem away from the boundary. (If X is a polygon, the Voronoi diagram is not a local object because each point must see neighbors at $O(1)$ distances.) The method of [23] uses cells also, but combines them with an incremental construction of the Delaunay triangulation. It is perhaps the closest in spirit to the uniform method of this paper, though still quite dissimilar.

3 A Uniform Cell Method

Before discussing the construction of the Delaunay triangulation, we must specify how it is to be stored. There are many possibilities; we choose a storage scheme which requires more than the minimum space necessary but makes it easy to work with the resulting triangulation. We store a triangulation by giving two integer arrays, `itt` and `itp`, in addition to the two real arrays needed to store the coordinates x_i and y_i of the points in X. (These arrays are named in accordance with a general scheme used throughout this work. An array beginning with `i` is a set of pointers, `t` stands for "triangle," and `p` stands for "point." Thus `itp` is a triangle-to point pointer array.) Let N_T be the number of triangles in the Delaunay triangulation. (It is a well-known consequence of Euler's formula that $N_T \leq 2N$, which simplifies the assignment of storage considerably.) Then the first integer array $\texttt{itp}(i,j)$, for $i = 1,2,3$ and $j = 1,2,\ldots,N_T$, points from triangles T_j of the Delaunay triangulation to points of X. Thus $k = \texttt{itp}(i,j)$ is the index of the ith vertex x_k of triangle T_j. (The vertices are numbered in counterclockwise order, with the sides numbered by their first vertex.) The second array $\texttt{itt}(i,j)$ points from triangles to neighboring triangles. Thus $k = \texttt{itt}(i,j)$, for $i = 1,2,3$ and $j = 1,2,\ldots,N_T$, is the index of the triangle T_k which lies across edge i of triangle T_j. If edge i of triangle T_j lies on the convex hull of X, there will be no neighboring triangle on that edge; we signal this situation by setting $\texttt{itt}(i,j) = 0$.

Given these two arrays, a total of $6N_T \leq 12N$ integer storage spaces, the triangulation can be used effectively. The triangulation is of course specified by the first array alone, but in most practical computations with a triangulation one needs to know the neighbors of a given triangle. For example, the standard way [21] of finding the triangle to which a point x belongs is to start with some arbitrary triangle and walk from neighbor to neighbor in the direction of x.

3.1 McLain's Method

Next we describe an algorithm due to McLain[24] which constructs the Delaunay triangulation step by step. It starts with a triangle belonging to the Delaunay triangulation and adds triangles one at a time until done, using the circumcircle criterion described below.

The first point, say x_i, is chosen at random. Then the second vertex of the first triangle, say x_j, is chosen from the set of closest points to x_i in X. The third vertex x_k of triangle 1 is chosen by the circumcircle criterion, applied to one side at a time of the edge $\overline{x_i x_j}$. This criterion says that we select the next vertex x_k to an edge $\overline{x_i x_j}$ having outward normal n so that a) x_k lies outside the edge and b) no other point of X lies in the interior of the circumcircle of the resulting triangle. This means that x_k is one of the minimizers of the signed distance of the center of the circumcircle from the line through x_i and x_j, counted positive on the side to which n points. The signed distance can be computed explicitly by analytic geometry; it is given by

$$t(x) = \frac{(x - x_i) \cdot (x - x_j)}{2(x - m) \cdot n}$$

where $m = (x_i + x_j)/2$ is the midpoint of $\overline{x_i x_j}$ and $\cdot$ is the dot product. If there are several points where $t(x)$ attains its minimum, any one of them may be chosen as the third vertex of the first triangle.

We now have the first triangle. We store the indices of x_i, x_j and x_k in the array $\texttt{itp}(m, 1)$, and set $\texttt{itt}(m, n) = -1$ initially for $1 \leq m \leq 3$ and $1 \leq n \leq 2N$. (We also make sure $x_i x_j x_k$ is oriented counterclockwise, and switch two points if necessary.)

The triangulation is completed by adding one triangle at a time—each triangle belongs to the final Delaunay triangulation. We loop through the indices n of existing triangles, adding a triangle (if possible) to each side i of triangle n which is not already shared by another triangle. It may be that it is impossible to add a triangle to an unoccupied edge, because there are no points of X outside the line extending that edge. In that case, we mark the edge as a boundary edge of the convex hull of X by setting $\texttt{itt}(m, n) = 0$, and proceed to the next edge. If there are points outside the current edge, on the other hand, we find the third vertex of the new triangle by the circumcircle criterion. Thus we find all minimizers of $t(x)$ over X which lie outside the current edge. If the minimizer is unique, as it usually is, it is taken as the third vertex of the new triangle. Otherwise, if there is more than one minimizer, then there are four or more cocircular points in X; namely the two ends of the current edge and the minimizers of $t(x)$. In this case, careless selection of the third edge can lead to degeneracy. We then search through all previous triangles for those triangles having cocircular vertices; those are checked for degeneracy against each prospective new triangle, and a nondegenerate choice of the third vertex is made. (This treatment of cocircularity represents a slight deviation from McLain's original algorithm,

which triangulated all cocircular vertices at this point, for a slight gain in efficiency.)

Whether there was a unique minimizer or not, we have now found the third vertex of a new triangle which can be added to the current edge. We add the new triangle to `itt` as a neighbor of the current triangle and vice versa, and add the three vertices to the next empty location in `itp`.

One further check must be made. The new triangle may well be a neighbor of some previously constructed triangle which we have not yet accounted for. Ignoring this possibility would lead to duplicate entries for the same triangle and thus to degeneracy of the triangulation. Hence we must check all previous triangles to find neighbors of the new triangle. If any are found, the appropriate entries must be made in `itt`.

This concludes the addition of a new triangle to the current data structure. We now proceed to the next unoccupied edge and repeat. When we run out of unoccupied edges, the Delaunay triangulation will be complete.

3.2 The Cell Method

McLain's method as presented in §3.1 is robust and easy to program, but can be quite slow when N is large. To speed it up, we introduce a cell structure and point-to-triangle pointers. Cells were used in [5, 23, 25] to speed up Voronoi diagram calculations. The basic idea is that only nearby points can affect the addition of a new triangle, if the points are arranged in a reasonably uniform way (so that the triangles don't get too long and thin). Thus we can organize the points into a data structure by spatial location and eliminate the necessity of searching through all N points of X every time we add a triangle. Of course, we have to make sure we get the right answer; we can't consider only nearby points without checking that we have included all the points which matter. Fortunately, the circumcircle criterion lends itself to such a check. Let C be the circle produced by minimizing $t(x)$ over a subset of X, with center $x_c = m + t_{min}n$ and radius $R = ||x_c - x_i||$, say. Then no point outside C can be the global minimizer of $t(x)$ over the whole set X. Thus any candidate for a new vertex excludes all points of X except those which lie in the intersection of a circle and a half-space. (Points on the wrong side of the edge are excluded as well as those outside the circumcircle.)

There are two stages of the triangle addition process which require checking $O(N)$ data. First, we have to find the minimizer of $t(x)$ among the $N-2$

remaining points of X. Second, when a new triangle is found, we have to check all previously found triangles to find those sharing an edge with the new triangle.

We reduce the cost of the minimization step by organizing the points of X into a data structure according to their spatial location. To do this, we first find the maximum and minimum x and y values, so that all points (x_i, y_i) lie in a rectangle B with sides parallel to the coordinate axes. Then we subdivide B into $N_B = O(\sqrt{N}) \times O(\sqrt{N})$ rectangular boxes and store each x_i in the box where it lies. To do this, we use an array `ibp` of length N which contains the index of each point and an array `ibp1` of length N_B which contains, in its jth location, the index in `ibp` where storage for the points in box j begins. Thus the points x_j in box i have their indices j stored in `ibp` between addresses `ibp1`(i) and `ibp1`$(i+1)-1$ inclusive. (The boxes i are ordered lexicographically and `ibp1`(N_B+1) points to the first empty space at the end of `ibp`.) This data structure can be constructed in three steps. First, we loop through the N points x_i, calculating which box j each x_i lies in and incrementing `ibp1`(j) by 1. (It is set to zero initially.) We now know how many points lie in each box i and thus how much storage to assign to the ith box in the array `ibp`. Second, we loop again through the points x_i. This time, we actually store the index i of x_i in the position in `ibp` where it belongs, keeping track of the next empty address for that box in `ibp1`. Third, we reset each `ibp1`(i) to indicate the beginning of storage in `ibp` for the points in box i, and this completes the construction of the uniform cell data structure.

Once this data structure has been constructed, we use it to reduce the cost of the first step in adding a triangle—minimizing $t(x)$—as follows. Say we are finding minimizers of $t(x)$ on a certain side of the segment $\overline{x_i x_j}$, indicated by the normal n. Find the boxes i_1 and i_2 which contain x_i and x_j (probably $i_1 = i_2$) and construct the smallest rectangular union C of boxes in the cell structure which contains both i_1 and i_2. Rather than minimizing $t(x)$ over all points, we now find only those minimizers of $t(x)$ which lie in C. Thus we only compute $t(x_k)$ for $x_k \in C$: to do this, we run through the boxes i which constitute C. For each such i, we run from $j =$ `ibp1`(i) to $j =$ `ibp1`$(i+1)-1$, and compute $t(x_k)$ where $k =$ `ibp`(j).

It is possible that C contains no points on the right side of $\overline{x_i x_j}$. If this happens (it rarely does), we search those points of X in boxes intersecting the correct side of $\overline{x_i x_j}$ according to the standard procedure for the $O(N^2)$ method.

If, on the other hand, there is at least one point in C on the correct side of $\overline{x_i x_j}$, then we will find at least one point x_k which minimizes $t(x)$ among all the points of X in C. This point may not be the global minimizer we are looking for, though usually it will be (if the Delaunay triangulation does not contain excessively many long and thin triangles), because C may not include the point we are really looking for. However, if we construct the circumcircle passing through x_i, x_j and x_k, we are guaranteed that *any* minimizer of $t(x)$ over all N points of X will lie inside the circumcircle. This follows from the definition of $t(x)$. In practice, the minimizer of $t(x)$ over x lying in C will be the global minimizer almost all the time, if the point distribution is reasonably uniform.

Hence if the circumcircle of x_i, x_j and x_k is contained in the original search area C, we have already found the minimizer of $t(x)$ over X. Otherwise, we expand C until it contains the circumcircle, and search the new union of boxes. This is guaranteed to produce all minimizers of $t(x)$ on the correct side of $\overline{x_i x_j}$.

Now if there is only one minimizer, we can take it as the third vertex of the new triangle, add the new triangle to our data structure, and proceed. This will almost always be the case for random points, but often fails to be true in the degenerate case when some of the points lie exactly on a rectangular grid. If there are several minimizers, degeneracy of the triangulation could result from a bad choice, so we have to choose the new vertex carefully among the minimizers. The idea is to avoid having the new triangle cross any previous triangle. A moment's thought shows that the new triangle can cross only triangles which have all three vertices on the circumcircle of x_i, x_j and the minimizers. To check these triangles efficiently, we use an array of pointers from the points of X to each triangle having them as a vertex This requires $3N_T \leq 6N$ integer storage locations, because each triangle has three verticesd and there are $N_T \leq 2N$ triangles, but each point belongs to six triangles only on the average; in the worst case, one point can belong to all N_T triangles. Hence the storage scheme must allow for variations in the length of triangle storage, from point to point. Also, this structure must be constructed along with the triangulation, dynamically, rather than all at once. Thus the scheme we used to construct the cell structure, which requires two sweeps over the points, cannot be applied here.

A similar situation is handled in [18] by the use of a *heap*; an array `ipt` of length about $9N$ is used as free-form storage, as we do for the pointers from boxes to points, with pointers `ipt1` and `ipt2` to the beginning and end of storage for indices of triangles to which a given vertex point belongs. When

a triangle is added to the list for a given point, a new copy of the list is made at the end of the heap and the old list is flagged for removal. When the storage space available is exhausted, garbage collection must be carried out to remove superseded lists.

Early implementations of our algorithm also used a heap, but we found that nonuniform point distributions required too much garbage collection. A *linked list* requires $6N_T$ memory, almost the same as a heap when the beginning and end pointers are taken into account, and no garbage collection, so our later implementations used a linked list instead. We use a single long array $\texttt{ipt}(i,j)$, where i runs from 1 to 2 and j from 1 to $3N_T$, structured as follows. The triangle indices for a given point are stored in a chain of non-contiguous locations, with the triangle index stored in $\texttt{ipt}(1,j)$ and $\texttt{ipt}(2,j)$ occupied by a pointer to the next triangle index. To get started, we have the first triangle x_j belongs to stored in $\texttt{ipt}(1,j)$ for $1 \le j \le N$; then we make a slight variation on the usual linked list by having $\texttt{ipt}(2,j)$ point to the place in ipt where the index of the *last* triangle (in order of creation) to which x_j belongs is stored. If this location is k, then $\texttt{ipt}(1,k)$ is the last triangle to which x_j belongs and $\texttt{ipt}(2,k)$ is the location in ipt where the *next to last* triangle for x_j is stored. The storage proceeds backwards in this way until the end of the triangle list for the jth point is signaled by a -1 in $\texttt{ipt}(2,n)$ for some n. Explicitly, the indices of the triangles for which x_j is a vertex are stored in locations $(1,j),(1,j_2 = \texttt{ipt}(2,j)),(1,j_3 = \texttt{ipt}(2,j_2)),(1,j_4 = \texttt{ipt}(2,j_3)),...$, until a -1 is encountered in $\texttt{ipt}(2,j_8)$, for example. Storing the first triangle in location j saves having pointers to the beginning of the list for each point, while storing the triangles backwards avoids the necessity of searching all the way through the list in order to add a triangle at its end. We can add a triangle to the list of a given point x_j simply by breaking and resetting the end link $\texttt{ipt}(2,j)$and adding the triangle to the next empty location at the end of ipt.

Given this storage arrangement, we can easily look up all triangles having x_k as a vertex, check if all three vertices lie on the circumcircle, and check for degeneracy if necessary. The degeneracy check is carried out by ensuring that the new triangle with vertices x_i, x_j and x_k does not separate, with its edges, the vertices of any previously constructed triangles. Once this test is passed, by choosing another minimizer if necessary, we have found the third vertex, and can proceed to add the new triangle to the existing data structure.

Now we must speed up the second $O(N)$ stage of the triangle addition process; we must check all previously constructed triangles and find those sharing a common edge with the new triangle. When we find them, we must

add the new triangle to their neighbor list `itt` and add them to the `itt` entry for the new triangle. This is easy to speed up, because we have introduced the linked list `ipt` which points from points to triangles having them as vertices; hence we can find all the desired triangles immediately in time proportional to their number and independent of N.

Finally, we update the pointers and proceed to the next edge of the growing triangulation. When there are no more edges to be augmented, the triangulation is concluded.

4 An Adaptive Cell Method

The uniform cell method is much more efficient than any quadratic method when N is large enough and the points are distributed in a reasonably uniform way. Unfortunately, in applications such as finite element triangulation, we do not want the points distributed uniformly. Even for the simple purpose of interpolation of a function known at irregularly distributed data points, we want to use more data points in regions where the function to be interpolated varies more rapidly [29]. Thus practical situations often lead to highly nonuniform point distributions. For these distributions, numerical experiments and theory both indicate that the uniform cell method runs in time close to its worst-case $O(N^2)$ timing. Even worse, the uniform method can be fooled simply by adding a few outlying points at a large distance from the majority of points; it will then construct a grid which is much too coarse, and the only remedy for this is adaptivity.

In this section, we present an *adaptive* cell method which runs much faster than the uniform method on certain nonuniform point distributions such as the normal. The basic idea is to sort points into boxes of varying size, so as to have no more than a fixed number s of points per box. This is done, as in [8, 12], by recursive bisection. The data structure used is described in §4.1. In §4.2, we describe our adaptive cell method, which differs from the uniform method both in the adaptive data structure and in the more complicated search strategy employed.

4.1 Adaptive Cells

In this section, we describe our data structure and how to construct and manipulate it. The object of the structure is to organize the points x_j spatially into groups of no more than say s points. This is done by recursively subdividing the rectangle B which contains X until no box contains more than s points. The boxes are then stored as follows.

At the end of the construction, we have partitioned B into N_B subboxes of varying sizes. For each box i, we store a) data on its spatial location and b) the indices j of the points x_j lying in box i. Part a) is achieved by storing three pointers per box, arranged in a $3 \times N_B$ array $\texttt{ibxy}(n, i)$; $L = \texttt{ibxy}(3, i)$ is the *level* of i in the sense that box i is 2^{-L} times smaller in each dimension than the original box B. Two more pointers $nx = \texttt{ibxy}(1, i)$

and $ny = \texttt{ibxy}(2,i)$ give the spatial location of the box, as if it were part of a regular grid on B composed entirely of boxes of level L; its lower left corner is at the point $(x = ax + nx \cdot hx, y = ay + ny \cdot hy)$. Here $B = [ax, bx] \times [ay, by]$ while the box sides of i have lengths $hx = 2^{-L}(bx - ax)$ and $hy = 2^{-L}(by - ay)$ respectively. Thus the array `ibxy` tells us the location and size of every box i from 1 to N_B. We have an overflow restriction $L \leq M$ where M is determined by the finite length of an integer in computer arithmetic; typically $M \geq 30$, so we can refine no more than 30 levels and the smallest box can be no more than $2^{-30} \approx 10^{-9}$ times smaller than the size of the domain. This has proved sufficient for most practical problems. Part b) is achieved by storing a list `ibp` of points lying in each box. Additional pointers `ibp1` and `ibp2` give the addresses in `ibp` of the beginning and end of the list of points in box i. Thus the points in box i have coordinates (x_j, y_j), where $j = \texttt{ibp}(k)$ for $k = \texttt{ibp1}(i), \ldots, \texttt{ibp2}(i)$.

The boxes are sorted lexicographically within each level, and arranged by level. Thus we use also a short array of pointers `ilb1` such that all the boxes on level L are given by $i = \texttt{ilb1}(L), \texttt{ilb1}(L)+1, \ldots, \texttt{ilb1}(L+1)-1$. Lexicographic ordering for boxes of the same size means that $\texttt{ibxy}(1,i) \leq \texttt{ibxy}(1,i+1)$ *and* if equality holds then $\texttt{ibxy}(2,i) < \texttt{ibxy}(2,i+1)$. Thus the boxes on each level are arranged from left to right into columns and within each column from bottom to top. The purpose of lexicographic ordering on each level is to speed up the operation of searching for a box with given values of $i1 = \texttt{ibxy}(1,i)$, $i2 = \texttt{ibxy}(2,i)$ and $i3 = \texttt{ibxy}(3,i)$; we simply carry out a binary search of $\texttt{ibxy}(1,i)$ and $\texttt{ibxy}(2,i)$ for i between $\texttt{ilb1}(i3)$ and $\texttt{ilb1}(i3+1)-1$. This operation is important when we construct the list of neighbors of a given box or when we find all boxes which intersect a given geometric object. This data structure is similar to that used in [8] and even more similar to that used in [12]. In the latter work, the three pointers were packed into a single 28-digit base-3 number, in a method designed for use with the vectorized bit-handling operations of the CDC CYBER 205. This limited the number of levels of subdivision possible to 16, which was sufficient for the calculations carried out in [12].

Next we describe the construction of the adaptive cell structure. We begin with the rectangle B and subdivide it into four boxes by bisecting each coordinate. We assign each point x_j to the box in which it lies. These boxes constitute level 1 of the structure. To construct level 2, we run through boxes created at level 1 and bisect any which contain more than s points, reassigning points to the subboxes in which they lie. The resulting boxes are added to the end of `ibxy`,`ibp`, `ibp1` and `ibp2` in the order in which

they were formed. Boxes which are subdivided are marked for deletion, and when the level 2 boxes have all been created, the subdivided boxes from level 1 are deleted and storage is reassigned. Thus empty boxes are kept but subdivided boxes are eliminated; the result is a partition of B into boxes with disjoint interiors. After deletion, pointers `ilb1` are made. The algorithm now proceeds recursively one level at a time. At each level, the boxes created in the previous level are subdivided where necessary, and the new boxes assigned numbers `ibxy` and storage in `ibp1` and `ibp2`. Subdivided boxes are deleted and storage moved up.

When this process terminates, either because the maximum number of levels is reached or more likely because no box has more than s points in it, the boxes on each level are sorted and rearranged in lexicographic order. Finally pointers `ipb` from points to boxes, showing which box a point lies in, are created, and we are done.

A typical cell structure obtained by this method is shown in Figure 1. To generate it, we took $N = 400$ points nonuniformly distributed in the unit square, and applied the algorithm just described, with no more than 3 points permitted per box. The construction of the adaptive structure turns out to require only a small fraction of the CPU time required for the whole triangulation process.

We need to carry out two primitive operations on this data structure. First, we consider the problem of finding the nearest neighbors of a given box i. The nearest neighbors, for our purposes, contain all boxes having a point in common with i, that is corner as well as side neighbors. To do this, we use the array `ibxy`. If all the boxes were the same size, the task would be easy; the spatial location numbers of the desired boxes would be obtained from `ibxy`(n,i) by adding 0, -1 or +1 to `ibxy`$(1,i)$ and `ibxy`$(2,i)$. A search through the boxes on level m_3 = `ibxy`$(3,i)$ would produce them and we would be done. Unfortunately, the boxes are not all the same size. Thus we must look on all levels for neighbors. Fortunately, the box numbers stored in `ibxy` are arranged to facilitate this. For example, suppose we are looking for the lower left corner neighbor of i. We begin on the same level as i by setting m_1 = `ibxy`$(1,i) - 1$ and m_2 = `ibxy`$(2,i) - 1$. These are the values `ibxy`$(1,j)$ and `ibxy`$(2,j)$ would have if a box j of the same size as i occupied the lower left corner position. Thus we search through boxes on level m_3 = `ibxy`$(3,i)$ for a box with numbers m_1 and m_2. If the search succeeds, we are done. If it fails, we must look for a larger or smaller box. A larger box is easier to find in general, so we go up first; set $m_1 \leftarrow m_1/2$, $m_2 \leftarrow m_2/2$ and $m_3 \leftarrow m_3 - 1$. (We use the FORTRAN integer divide, which

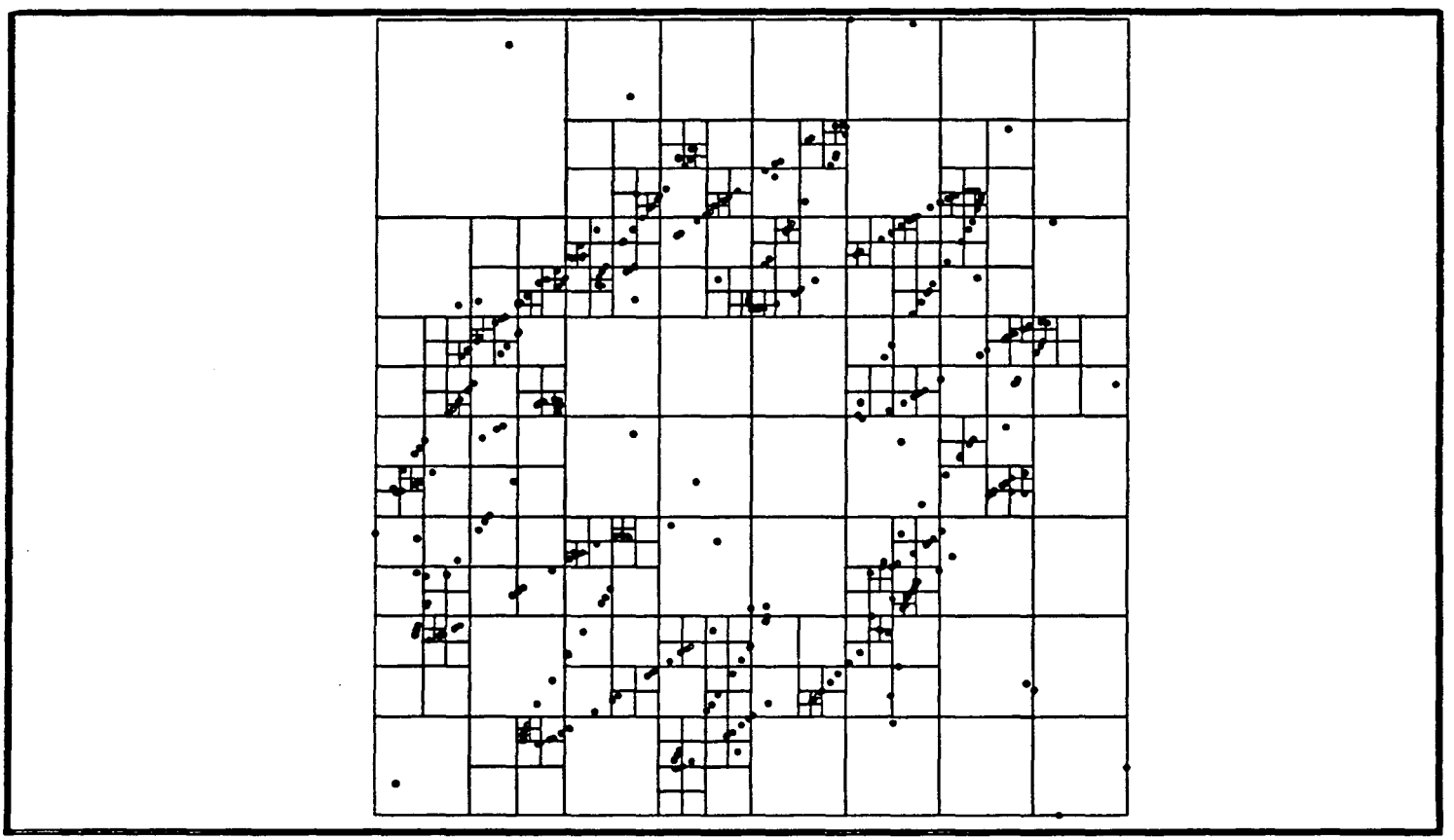


Figure 1: Adaptive cell structure for $N = 400$ nonuniformly distributed points in the unit square, with no more than 3 points per box.

throws away the remainder.) These are the numbers a box one level larger in the lower left corner position would have, so we look on level m_3 for the parent having numbers m_1 and m_2. This procedure is repeated if necessary until either we find the box or we reach the top level without finding it. If the latter occurs, we have to look for a smaller box. The corners and sides differ here because on the corners we are looking for a single box, while on the sides we are looking for several smaller boxes. On the lower left corner, for example, we seek a smaller box by putting $m_1 \leftarrow 2 \cdot m_1 + 1$, $m_2 \leftarrow 2 \cdot m_2 + 1$, $m_3 \leftarrow m_3 + 1$, and searching on level m_3, then repeating this procedure as needed until the box is found. Before any searching at all is done, of course, we must take a precaution against going outside B; thus we check that m_1 and m_2 lie in the range $[0, 2^{m_3} - 1]$. If this constraint is violated, the box we are seeking does not exist and there is no neighbor on that side of box i.

On the sides, the search for smaller neighbors is slightly more complicated. We begin, say on the left side, with $m_1 \leftarrow m_1 - 1$ and $m_2 \leftarrow m_2$. If no box on level m_3 with numbers (m_1, m_2) exists, then we look for smaller neighbors, possibly several of them. First, we subdivide (m_1, m_2) into four boxes and put the right-hand two boxes on a stack. The left two boxes are discarded. We now run through the stack, searching for each box on the level where it should live. If it is found, it is added to the neighbor list and we continue with the next stack entry. If no such box exists, the box is subdivided, the right-hand two boxes are stacked and the left-hand ones discarded, and we continue with the next stack entry. When the stack is empty, this process terminates and we have the list of neighbors. If necessary, duplicates are discarded (a large box may be found several times) and we are done.

Another operation we need to carry out with this data structure is to find all boxes which intersect a given geometrical object Ω such as a square or (in our case) the intersection of a circle with a half-space. A straightforward and robust way to do this is to begin on the top level and search every level for every box intersecting Ω. A faster method uses recursion. We begin by stacking the four top-level boxes. Each is examined for existence and intersection; if it exists in our data structure and intersects Ω it is added to our list, if it exists and does not intersect it is discarded, and if it does not exist, then it is subdivided, its subboxes are stacked, and we proceed with the next item in the stack.

4.2 An Adaptive Cell Method

The adaptive cell method we now present is one of several possible straightforward extensions of the uniform method, so we concentrate on the differences.

The main difference is in the search strategy, because that is where the cell structure was used. Our adaptive search strategy for adding a triangle to an edge $\overline{x_i x_j}$ is as follows.

The first step is to search the box or the two boxes containing the endpoints x_i and x_j of the current edge. If a point x_k is found to minimize $t(x)$ over this search area, we compute the circle through x_i, x_j and x_k and test whether it is contained entirely within the search area. If it is, we have found the global minimizer and can proceed with the degeneracy check and the triangle addition precisely as in the uniform scheme. Otherwise, or if no point at all was found in the first search area on the outside of $\overline{x_i x_j}$, we must enlarge the search area.

Our next step is then to find the nearest neighbor boxes of the one or two boxes of the first search area and take their union as the second search area. Heuristically, we expect a layer of nearest neighbors to be sufficient in most cases because they will screen the current edge from distant points. The second search can again have three outcomes. First suppose no point has yet been found when the second search terminates. Then it is quite likely but not certain that $\overline{x_i x_j}$ is on the boundary of the convex hull of X; thus we find all boxes intersecting the half-space outside $\overline{x_i x_j}$ and take their union as the third search area. If a point has been found, on the other hand, then we have a local minimizer x_k. Let C be the circumcircle of x_i, x_j and x_k. If the interior of C is contained in the second search area, we have found the global minimizer and can proceed with the degeneracy check and so forth.

Otherwise, we must enlarge our scope to the third and final search area comprising all boxes which intersect the interior of C. After searching the third search area, we have either found all global minimizers of $t(x)$ which lie outside $\overline{x_i x_j}$, or determined that $\overline{x_i x_j}$ lies on the boundary of the convex hull of X, and can proceed with the degeneracy check and so forth.

A considerable speedup (usually about thirty percent), if enough memory is available, is obtained by precomputing all neighbors of nonempty boxes and storing them. This eliminates the necessity of repeatedly finding the neighbors of boxes, a considerable savings when the number of points per box is large. If there are ten points per box, then there are usually about

$N_B = N/5$ boxes, so the storage is probably available. It requires an array of length perhaps $10N_B \leq 2N$ and another of length N_B.

5 Numerical Results

We have implemented the three algorithms described in this paper in portable ANSI FORTRAN 77 and tested their performance on many sets of data points. We used a SUN SPARCstation 1+ with 64 megabytes of RAM and the SUN FORTRAN optimizer. For comparison purposes, this setup runs Linpack benchmarks at about 1.5 megaflops.

Results from two sets of test data will be reported here. The first set consisted of N points generated by a pseudorandom number generator, uniformly distributed on the region Ω interior to the circle with center (0.5,0.5) and radius 0.45 but exterior to the circle with center (0.45,0.3) and radius 0.25. The resulting Delaunay triangulation is shown in Figure 2 for the case $N = 800$.

The second set of test data were composed of four sets of $N/4$ normally distributed points, centered at four points in $[0,1]^2$ and with variances given by $\sigma = 0.15, 0.15/7, 0.15/7^2, 0.15/7^3$. The resulting Delaunay triangulation is shown in Figure 3 for the case $N = 800$.

Table 1 reports the results of triangulating the first set of data points, with N ranging from 100 to 51,200. The column headings have the following meanings;

N is the number of data points.

N_T is the number of triangles produced; N_T was checked against the well-known Euler formula [20] to ensure its consistency with the number of points and the number of boundary points.

T_q is the CPU time in seconds required by the quadratic algorithm of §3.1, estimated by extrapolation for $N > 10,000$ to avoid wasting too much computer time.

T_u is the CPU time required by the uniform cell method, with $N_B = (\lfloor\sqrt{N}\rfloor)^2$ cells.

T_a is the CPU time required by the adaptive cell method, using $s = 25$ as the maximum number of points per cell permitted.

Table 2 reports the results of triangulating the second set of data points, with N ranging from 100 to 204,800. The parameters have the same meaning as

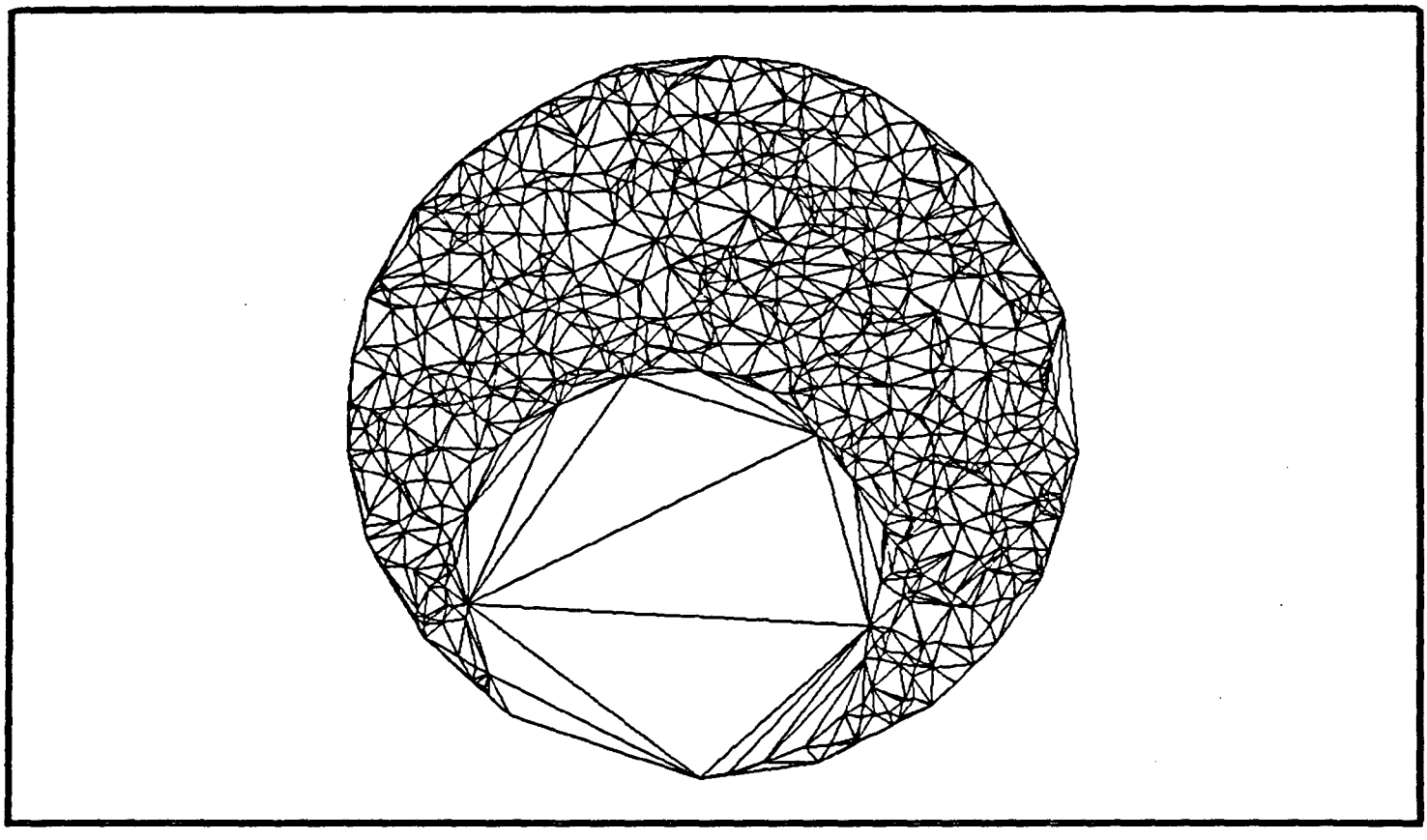


Figure 2: Delaunay triangulation of a set of $N = 800$ uniformly distributed random points between two circles

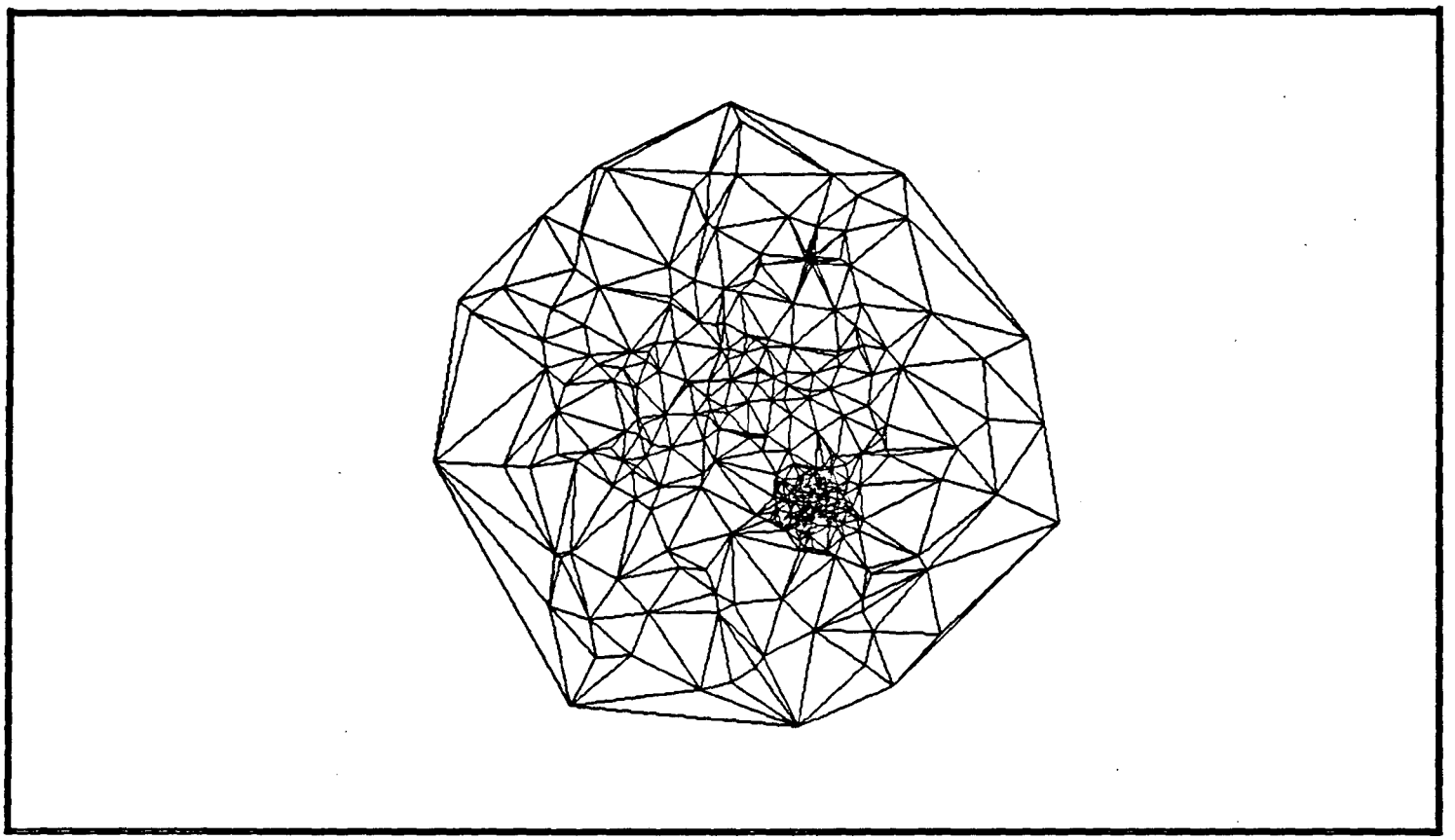


Figure 3: Delaunay triangulation of a set of $N = 800$ nonuniformly distributed random points.

in Table 1, except that T_u is estimated by extrapolation for $N > 20000$, to avoid using too much computer time. It is clearly growing quadratically by this point.

We can draw the following conclusions from these tables; first, both the uniform and adaptive methods are faster than the quadratic method as soon as $N \geq 200$. Thus for large problems, they are much to be preferred if sufficient memory is available. The uniform method requires about $26N$ integer memory in addition to $2N$ real storage for x and y; about $12N$ of the integer storage is used just to store the triangulation. Thus the uniform method uses only about twice the minimum amount of memory. The adaptive method typically has similar storage requirements, despite the larger amount of information it stores, because we take bigger boxes and hence have fewer of them. It is difficult to give a tight upper bound for its memory usage, especially when the number of points per box is chosen very small.

Second, the adaptive method is about twice as slow as the uniform method on uniform points. Some slowdown is probably unavoidable, because of the inherently greater overhead involved in an adaptive cell structure. We conjecture, however, that our implementation could be speeded up by a factor of two, by more arduous programming or by altering the search strategy slightly. We have not attempted to optimize the program extensively, concentrating instead on demonstrating that it achieves $O(N \log N)$ performance without worrying about a possible factor of two in the constant.

Third, on nonuniformly distributed points, the uniform method behaves well when N is small, but degenerates to $O(N^2)$ performance when N gets large. This is to be expected on theoretical grounds. The adaptive method, on the other hand, displays a gratifyingly regular $O(N \log N)$ performance throughout the whole range of N, It beats the uniform method consistently when $N \geq 400$, and outperforms the quadratic method as soon as $N \geq 200$. The CPU time required by the adaptive method is only increased by about ten percent by the nonuniformity of the point distribution.

Table 3 gives fuller statistics about the adaptive method applied to the second set of nonuniformly distributed data points, with N ranging from 100 to 204,800. We give the following information: N_B is the number of boxes created by the adaptive method. M/N is the integer storage required (in addition to the $24N$ integer storage required by all three methods), divided by N. L is the highest level used in construction of the adaptive cell structure. P_1 respectively P_2 is the percentage of the edges for which the initial respectively second search area had to be enlarged. Clearly many of the initial

search areas were enlarged, because only 25 points per box were used, but very few of the nearest-neighbor searches were unsuccessful. Tests with more points per box decreased the percentage of expansions of the first search area, but failed to improve the total running time, because each search then took longer. Note that the adaptive method requires only about $4N$ additional integer storage beyond that required by all three methods.

N	N_T	T_q	T_q/N^2	T_u	$T_u/N \log N$	T_a	$T_a/N \log N$
100	178	0.11	0.11E-04	0.06	0.13E-03	0.15	0.33E-03
200	378	0.42	0.10E-04	0.14	0.13E-03	0.34	0.32E-03
400	770	1.63	0.10E-04	0.31	0.13E-03	0.72	0.30E-03
800	1563	6.58	0.10E-04	0.66	0.12E-03	1.63	0.30E-03
1600	3156	26.21	0.10E-04	1.37	0.12E-03	3.54	0.30E-03
3200	6347	111.18	0.11E-04	2.88	0.11E-03	7.86	0.30E-03
6400	12731	438.06	0.11E-04	5.98	0.11E-03	16.11	0.29E-03
12800	25514	1752.23*	0.11E-04	12.52	0.10E-03	35.30	0.29E-03
25600	51092	7008.94*	0.11E-04	27.96	0.11E-03	68.92	0.27E-03
51200	102260	28035.75*	0.11E-04	62.32	0.11E-03	144.64	0.26E-03

Table 1: Timings for constructing the Delaunay triangulation of N uniformly distributed points in a region between two circles, using the quadratic (T_q), uniform cell (T_u) and adaptive cell (T_a) methods. N_T is the number of triangles in the triangulation. Asterisks denote timings obtained by extrapolation for the quadratic method.

N	N_T	T_q	T_q/N^2	T_u	$T_u/N \log N$	T_a	$T_a/N \log N$
100	189	0.14	0.14E-04	0.11	0.24E-03	0.20	0.43E-03
200	387	0.54	0.13E-04	0.32	0.30E-03	0.44	0.42E-03
400	789	2.12	0.13E-04	1.07	0.45E-03	0.97	0.40E-03
800	1586	8.75	0.14E-04	3.86	0.72E-03	2.06	0.39E-03
1600	3184	33.73	0.13E-04	13.35	0.11E-02	4.42	0.37E-03
3200	6385	136.05	0.13E-04	51.04	0.20E-02	9.22	0.36E-03
6400	12789	566.79	0.14E-04	198.19	0.35E-02	19.66	0.35E-03
12800	25588	2267.16*	0.14E-04	779.70	0.64E-02	40.95	0.34E-03
25600	51186	9068.64*	0.14E-04	3118.80*	0.12E-01	82.94	0.32E-03
51200	102385	36274.56*	0.14E-04	12475.20*	0.22E-01	169.57	0.31E-03
102400	204788	145098.25*	0.14E-04	49900.80*	0.42E-01	350.75	0.30E-03
204800	409587	580393.00*	0.14E-04	199603.22*	0.80E-01	716.35	0.29E-03

Table 2: Timings for constructing the Delaunay triangulation of N nonuniformly distributed points, using the quadratic (T_q), uniform cell (T_u) and adaptive cell (T_a) methods. N_T is the number of triangles in the triangulation. Asterisks denote timings obtained by extrapolation for the quadratic and uniform methods.

N	N_T	N_B	M/N	L	P_1	P_2	T	$T/N \log N$
100	189	28	10.02	8	62.94	9.64	0.22	0.48E-03
200	387	40	7.24	10	69.02	5.04	0.45	0.42E-03
400	789	58	5.60	10	68.88	2.63	1.03	0.43E-03
800	1586	109	5.03	11	66.69	1.88	2.10	0.39E-03
1600	3184	178	4.42	12	66.53	0.75	4.44	0.38E-03
3200	6385	319	4.12	13	67.11	0.33	9.46	0.37E-03
6400	12789	583	3.91	14	65.40	0.14	19.71	0.35E-03
12800	25588	1144	3.87	14	65.39	0.13	40.68	0.34E-03
25600	51186	2275	3.85	15	65.30	0.04	82.82	0.32E-03
51200	102385	4426	3.80	16	64.65	0.03	172.10	0.31E-03
102400	204788	9031	3.84	16	65.24	0.02	348.83	0.30E-03
204800	409587	17689	3.80	17	64.89	0.01	721.99	0.29E-03

Table 3: Information on the adaptive method for the second test case. N_B is the number of boxes used. M/N is the additional memory used by the adaptive method with neighbor lists stored, divided by N. L is the highest level used in the adaptive box structure. P_1 and P_2 are the percentages of the first and second search areas which had to be enlarged after the search. T is the CPU time required, in seconds.

6 Refinements and Generalizations

There are many possible ways to refine and generalize our methods. As noted in [5], uniform cell methods in general seem likely to be useful in many local problems of computational geometry. Their worst-case behavior is usually far from optimal, but their average behavior is quite fast when the data is uniformly distributed. The adaptive method presented here attempts to extend the cell idea to be useful in non-uniform situations.

There are many details of the adaptive method which could be implemented differently. Currently, we search only the nearest neighbor boxes, then use the resulting information to construct a new search area in which the third vertex is guaranteed to lie. Another possibility is to expand the search area one shell at a time, by adding all boxes which are neighbors of boxes already searched, and checking the circumcircle inclusion after each shell is searched. This version seems much more complicated to program., and many boxes must be found and added to the list when the number of shells grows beyond one or two. Thus this approach seemed likely to be slower in practice, even though it may offer a better chance at proving the method optimal in theory. Thus it was not implemented.

The adaptive method is fast when N is very large and X is highly non-uniform, but there are limits to the nonuniformities it can handle. Its theoretical worst-case performance seems quite difficult to analyze. Experiments, however, indicate that our implementation runs quite slowly in the (rather pathological) case when the points of X almost all lie on a curve. This is because the Delaunay triangulation connects points from one side of the curve to the other with very long thin triangles; this destroys the *locality* of the problem, because many points have to see vertices at $O(1)$ distance independent of N. See Figure 1 and the Delaunay triangulation of that set of points, superimposed on the adaptive cell structure in Figure 4. Clearly many points have to reach outside their nearest neighbors in order to build the Delaunay triangulation, making cell methods infeasible. This case can be excluded if we analyze the expected time required when the points are randomly but not quasi-uniformly distributed in the sense of [5], but the Delaunay triangulation nevertheless has triangles of bounded aspect ratio. A $N \log N$ expected time seems likely in this case, though quite difficult to prove. Something like an *a priori* estimate seems to be needed here, perhaps bounding the aspect ratio of the Delaunay triangulation in terms of an approximate Hausdorff dimension of the set X; this is an area of research still in its infancy, though such ideas were used in [35].

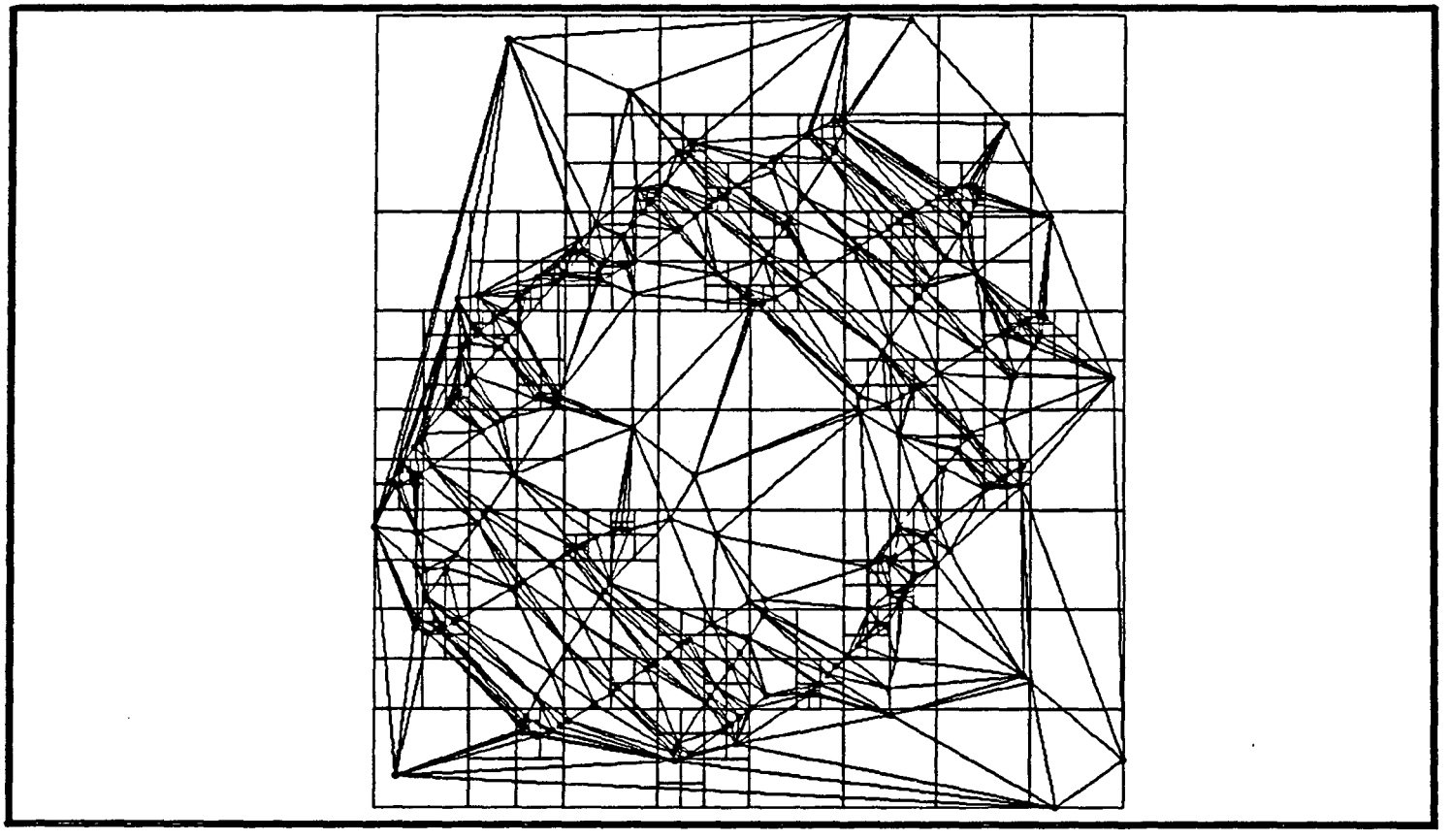


Figure 4: Delaunay triangulation of a set of $N = 400$ non-uniformly distributed points in the unit box. The nonlocality of the problem is evident in the many long thin triangles reaching beyond their nearest neighbor boxes.

Another problem which can be solved by our methods is constrained Delaunay triangulation [10]. In this problem, one must triangulate a point set X subject to the restrictions that a set E of edges be included and that no edge may cross any barrier from a set B of line segment (or more complicated) barriers. The locality of this problem implies that our cell methods extend immediately to handle it as well.

More generally, our methods seem likely to be useful in other geometric problems having a local character. Nearest neighbor search is an obvious though simple example. It can be solved very efficiently with a spiral search through our adaptive cell structure.

Another important generalization is the construction of the three-dimensional Delaunay tessellation. Our method generalizes straightforwardly in every detail, unlike many other two-dimensional fast algorithms which cannot be extended to three dimensions. The Delaunay tessellation in three dimensions can be expensive to compute, because the number of tetrahedra can be as large as $O(N^2)$ rather than $O(N)$. However, the adaptive cell structure still requires only $O(N \log N)$ storage and time to construct, making it relatively even less expensive than in the two-dimensional case.

7 Conclusions

We have presented fast algorithms for constructing the Delaunay triangulation in the plane and described numerical experiments, on up to two hundred thousand points, which indicate the efficiency of these new methods.

The uniform cell method outpaces the straightforward quadratic method as soon as we have more than a hundred points to triangulate, even when the points are non-uniformly distributed. It slows down for non-uniform points, to become asymptotically $O(N^2)$ for normally distributed points (though still several times faster than the quadratic method in our tests).

Thus we introduce an adaptive cell method which displays an $O(N \log N)$ behavior even on nonuniformly distributed points. It costs more overhead than the uniform method, making it slightly less efficient on uniform points and generally when N is less than about 400, but it is several hundred times faster for two hundred thousand nonuniformly distributed points, a substantial improvement.

8 Acknowledgements

I would like to thank B. Chazelle, A. Chorin and G. Russo for helpful conversations.

References

[1] H. AKIMA, A method of bivariate interpolation and smooth surface fitting for irregularly distributed data points. *ACM Trans. Math. Softw.* **4**, 148-164 (1978).

[2] F. ALMGREN, "Computing soap films and crystals," in *Computing Optimal Geometries*, edited by J. E. Taylor (American Mathematical Society, 1991).

[3] F. ALMGREN, "The geometric calculus of variations and modelling natural phenomena," in *Statistical Geometry and Differential Geometry*, edited by H. T. Davis and J. C. C. Nitsche (IMA Volumes in Mathematics and its Applications, Springer-Verlag, New York, 1991).

[4] F. AURENHAMMER, "Voronoi Diagrams—A Survey," Technical Report, Gras Technical University, Austria.

[5] J. L. BENTLEY, B. W. WEIDE AND A. C. YAO, Optimal expected-time algorithms for closest-point problems. *ACM Trans. Math. Softw.* **6**, 563-580 (1980).

[6] C. BORGERS AND C. S. PESKIN, A Lagrangian fractional step method for the incompressible Navier-Stokes equations on a periodic domain. *J. Comput. Phys.* **70**, 397-??? (1987).

[7] A. BOWYER, Computing Dirichlet tessellations. *The Computer J.* **24**, 162-166 (1981).

[8] J. CARRIER, L. GREENGARD AND V. ROKHLIN, A fast adaptive multipole algorithm for particle simulations. *SIAM J. Sci. Stat. Comput.* **9**, 669-686 (1988).

[9] A. K. CLINE AND R. J. RENKA, A storage-efficient method for construction of a Thiessen triangulation. *Rocky Mountain J. Math.* **14**, 119-139 (1984).

[10] A. K. CLINE AND R. J. RENKA, A constrained two-dimensional triangulation problem and the solution of closest node problems in the presence of barriers. *SIAM J. Num. Analysis* **27**, 1305-1321 (1990).

[11] H. H. DANNELONGUE AND P. A. TANGUY, Efficient data structures for adaptive remeshing with the FEM. *J. Comput. Phys.* **91**, 94-109 (1990).

[12] L. VAN DOMMELEN AND E. A. RUNDENSTEINER, Fast adaptive summation of point forces in the two-dimensional Poisson equation. *J. Comput. Phys.* **83**, 126-147 (1989).

[13] R. A. DWYER, A faster divide-and-conquer algorithm for constructing Delaunay triangulations. *Algorithmica* **2**, 151 (1987).

[14] S. FORTUNE, A sweepline algorithm for Voronoi diagrams. *Algorithmica* **2**, 137 (1987).

[15] M. J. FRITTS, "Three-dimensional algorithms for grid restructuring in free-Lagrangian calculations," in *The Free Lagrange Method, Lecture Notes in Physics 238,*, edited by M. J. Fritts, W. P. Crowley and H. Trease (Springer-Verlag, Berlin, 1985).

[16] M. J. FRITTS AND J. P. BORIS, The Lagrangian treatment of transient problems in hydrodynamics using a triangular mesh. *J. Comput. Phys.* **3**, 173-215 (1979).

[17] D. E. FYFE, E. S. ORAN AND M. J. FRITTS, Surface tension and viscosity with Lagrangian hydrodynamics on a triangular mesh, *J. Comput. Phys.* **76**, 349-384 (1984).

[18] P. J. GREEN AND R. SIBSON, Computing Dirichlet tesselations in the plane. *The Computer J.* **21**, 168-173 (1978).

[19] A. JAMESON AND T. J. BAKER, "Euler calculations for a complete aircraft," in *Proceedings International Conference on Numerical Methods in Fluid Dynamics, Lecture Notes in Physics 264,* edited by ??? (Springer-Verlag, Berlin, 1987).

[20] C. LAWSON, "Software for C^1 surface interpolation," in *Mathematical Software III*, edited by J. Rice (Academic Press, New York, 1977).

[21] D. T. LEE AND B. J. SCHACHTER, Two algorithms for constructing a Delaunay triangulation. *Int. J. Comput. Inf. Sci.* **9**, 219-242 (1980).

[22] B. A. LEWIS AND J. S. ROBINSON, Triangulation of planar regions with applications. *The Computer J.* **21**, 324-332 (1978).

[23] A. MAUS, Delaunay triangulation and the convex hull of n points in expected linear time. *BIT* **24**, 151-163 (1984).

[24] D. H. MCLAIN, Two dimensional interpolation from random data. *The Computer J.* **19**, 178-181 (1976).

[25] T. OHYA, M. IRI AND K. MUROTA, A fast Voronoi diagram algorithm with quaternary tree bucketing. *Inf. Process. Lett.* **18**, 178-181 (1984).

[26] C. PESKIN, *J. Comput. Phys.* **25**, 220 (1977).

[27] F. P. PREPARATA AND M. I. SHAMOS, Computational Geometry: An Introduction (Springer-Verlag, New York, 1985).

[28] R. J. RENKA, Algorithm 624: Triangulation and interpolation at arbitrarily distributed points in the plane. *ACM Trans. Math. Softw.* **10**, 440-442 (1984).

[29] R. J. RENKA AND A. K. CLINE, A triangle-based C^1 interpolation method. *Rocky Mountain J. Math.* **14**, 119-139 (1984).

[30] G. RUSSO, Deterministic diffusion of particles. *Comm. Pure Appl. Math.*, to appear.

[31] M. I. SHAMOS AND D. HOEY, "Closest-point problems," in *Proceedings 16th IEEE Symposium on Foundations of Computer Science, October 1975*, p. 151.

[32] R. SIBSON, Locally equiangular triangulations. *The Computer J.* **21**, 243-245 (1978).

[33] W. D. SMITH, *Studies in computational geometry motivated by mesh generation* (Ph.D. Thesis, Princeton University Department of Computer Science, 1989).

[34] J. STRAIN, A Delaunay method for crystal growth, in preparation.

[35] J. STRAIN, Fast potential theory II: Layer potentials and discrete sums. submitted to *J. Comput. Phys.*

[36] M. TANEMURA, T. OGAWA AND N. OGITA, A new algorithm for three-dimensional Voronoi tessellation. *J. Comput. Phys.* **51**, 191-207 (1983).

[37] D. F. WATSON, Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes. *The Computer J.* **24**, 167-172 (1981).

[38] F. W. WILSON, R. K. GOODRICH AND W. SPRATTE, Lawson's algorithm is nearly optimal for controlling error bounds. *SIAM J. Numer. Anal.* **27**, 190-197 (1990).

[39] M. A. YERRY AND M. S. SHEPHARD, Automatic three-dimensional mesh generation by the modified-octree technique. *Int. J. Num. Meth. Engrg.* **20**, 1965-1990 (1984).

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
INFORMATION RESOURCES DEPARTMENT
BERKELEY, CALIFORNIA 94720