SIAM J. SCI. STAT. COMPUT.
Vol. 7, No. 3, July 1986

DATA STRUCTURES FOR ADAPTIVE GRID GENERATION*

MARSHA J. BERGER†

Abstract. This paper describes data structures and algorithms for the automatic generation of adaptive subgrids, a technique used with adaptive mesh refinement for solving partial differential equations. Our algorithms generate a nested sequence of finer and finer grids on an underlying coarse grid. There are two aspects to the data structures. Trees are used to do the grid management for this type of grid structure. Secondly, the automatic grid generation algorithms use data structures with special nearest neighbor properties. Examples of grids from actual adaptive numerical computations are shown.

Key words. adaptive grid generation, data structures

AMS(MOS) subject classifications. 65M99, 68G10

1. Introduction.

In this paper we discuss the use of data structures in tackling an adaptive grid generation problem. These grids are to be used in the solution of partial differential equations (pdes) by finite difference schemes using adaptive mesh refinement. In this approach, a nested sequence of locally uniform fine grids is superimposed on an underlying coarse grid until a given accuracy criterion is attained. These methods were originally developed for the solution of hyperbolic pdes (see [3], [7]), but we believe that our algorithms and data structures have a wider applicability. For example, we have extended our adaptive hyperbolic method to handle steady-state transonic flow [5]. Even in this steady-state case, the grid management methods need to be dynamic, however, since the regions needing refinement (for example the shock location) are not known in advance. Our grid generation package is also currently being used in a mesh refinement program for solving elliptic pdes [8]. It is natural to use these adaptive grids in conjunction with multigrid methods [16]. Indeed, we have begun such work for the steady Euler equations [6].

There are two aspects to the use of data structures in this context. The first is storing and manipulating grids. We use trees and linked lists to keep track of this irregular grid structure. The data structures themselves are not uncommon; it is their application to the numerical solution of pdes that is new. The second aspect is our method of automatic grid generation, which is based on data structures with special nearest neighbor properties. Our grid generation is at most a two pass algorithm which clusters grid points and fits locally regular rectangles for the fine grids.

Before describing our data structures, we justify our use of locally uniform, possibly rotated, rectangles as the building blocks of a general adaptive mesh refinement method. Rectangles have the simplest user interface. The same integrator for the coarse grid may be used to integrate all the fine grids too. By separating the integrator from the adaptivity strategy, an off-the-shelf integrator can be used without modification, as was done in the transonic flow calculations of [5]. This simplifies the programming for each new application and allows an easier front end. For rotated rectangular grids, it is easy to automatically transform the difference equations into the rotated coordinate system. If the area needing refinement is diagonal to the grid, a smaller total area is refined if we allow the rectangles to rotate too. Moreover, in some calculations it is numerically advantageous to use a coordinate system which is approximately locally normal and tangent to a front in the solution. Using multiple oriented subgrids allows

* Received by the editors March 22, 1983, and in final revised form May 20, 1985. This research was supported in part by the U.S. Department of Energy under contract DEAC0276ER03077-V.

† Courant Institute of Mathematical Sciences, New York University, New York, New York 10012.

this to happen. This would also permit refinement in only one coordinate direction. However, there is additional overhead associated with rotated grids, in both the interpolation procedures and the interface equations needed between the grids, particularly if the interface conditions need to be conservative. This overhead will be discussed later. Our algorithms can, therefore, be used to create rotated as well as nonrotated subgrids. For example, in Fig. 1a we show a coarse grid where the grid points which need refinement are marked with an X. Fig. 1b shows some typical fine grids our grid generation package produces in this situation. If we do not permit rotation, the subgrids our package produces are shown in Fig. 1c.

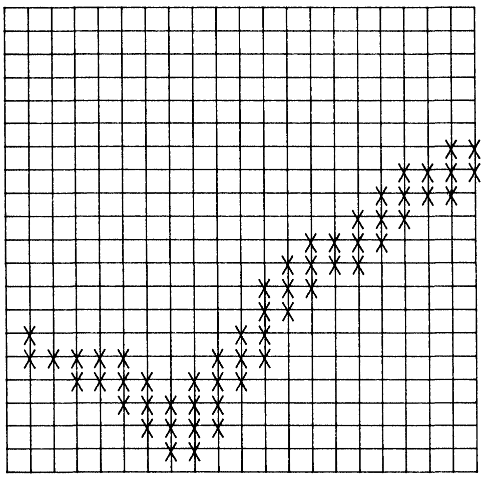


FIG. 1a. *Coarse grid with grid points needing refinement marked with an 'X'.*

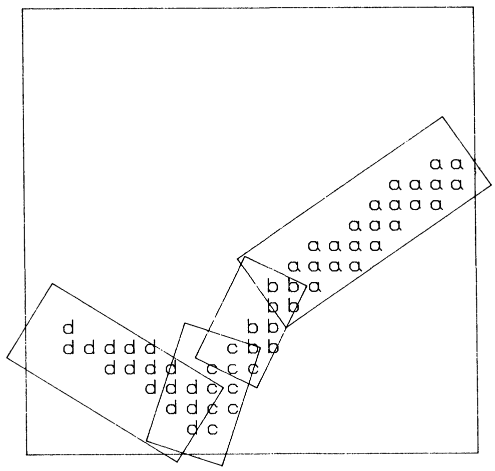


FIG. 1b. *Grid generation example using rotated rectangles for the fine grids that enclose the marked grid points.*

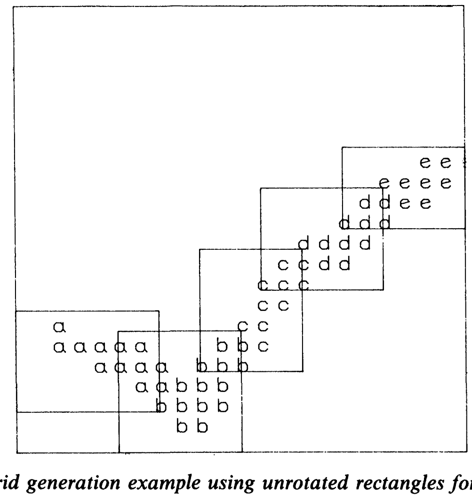


FIG. 1c. *Grid generation example using unrotated rectangles for the fine grids.*

In either case, this use of locally embedded fine grids should be contrasted with moving grid points, where regions of a grid are refined by "attracting" points into the region at the expense of resolution in the rest of the region (see [15] and references there). Such methods often have difficulty in controlling grid skewness, which can degrade accuracy. This will be even more of a problem for three-dimensional calculations. However, they do not have the problem of internal interfaces found in our method of refinement.

We wish to make two points about the grid structure. These fine grids are not merged into the underlying coarse grid, but are kept separately defined, each with its own solution storage vector. In this way, we take advantage of the local uniformity of each grid. The coarse grid points underneath a fine grid are in some sense wasted, unless they participate in the solution process itself, as in multigrid methods. However, even if this is not the case, we believe this is a smaller price to pay than alternatives which involve cell by cell or column oriented refinement and use much more storage overhead for pointers. Such methods typically have difficulties with multiply connected regions and inhibit the vectorization of integration algorithms.

The second point we make about our grid structure is the following. It may happen that many levels of refinement are needed to get sufficiently fine local resolution. This is no problem for the grid generation routines, only for the data structures that keep track of the grids. The grid generation algorithm is applied to the flagged points at each grid level to create the next finer level of grids. Fig. 2 illustrates an initial grid

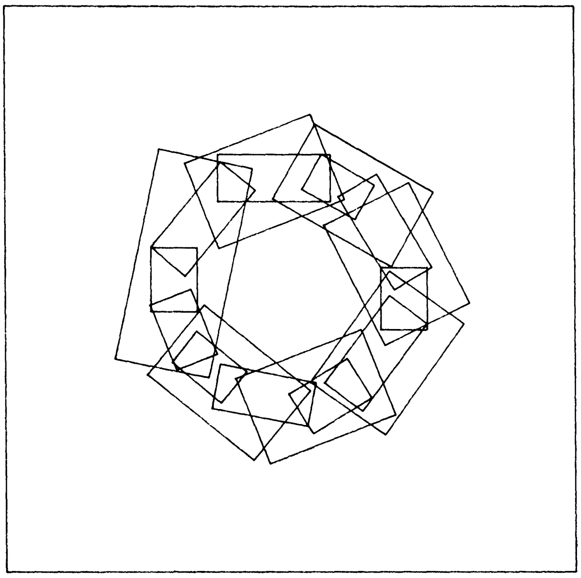


FIG. 2. *There are two levels of refinement around the circular expanding shock, inside the coarse grid.*

configuration with 3 levels of grids for a radially symmetric expanding shock problem. An error estimator applied to the coarse grid at level 0 yields 7 level 1 grids, each approximately 5 coarse mesh widths wide. When these grids have their error estimated, the grid generation algorithm yields 11 level 2 grids. Now, the level 2 grids are approximately 5 fine mesh widths wide. In this figure the reader can also see how one level 2 grid can be (partially) nested in one or more level 1 grids.

Given this nested hierarchy of finer and finer rectangular subgrids, we can now describe the two main roles that data structures play. In § 2, we describe the trees and lists used to store and access grid information in one and two space dimensions and to manage the one large array where the solution vector for each grid is stored. The more unusual way that data structures play a role is in the grid generation algorithms. Given a list of flagged grid points, any one of us can take a pen and draw in good grids by eye. It is not so easy to get the computer to do it. We have been able to draw on some ideas from the pattern recognition literature in designing our grid generation package. However, most of these scene analysis programs make many passes over the data from one scene. Since for a time-dependent pde our grid generation method will be used every few timesteps, we need a faster algorithm. In § 3, we describe our two-part grid generation algorithm and the graph structures on which the algorithm is based. This is a more important and difficult problem, and we believe that more work will yield methods substantially better than the preliminary heuristics discussed here.

2. Data structures for grid management.

In this section we describe the trees and linked lists used to store and access information for a grid structure. The one-dimensional structure is described first; a generalization of it is used in the two-dimensional case. We assume the reader is already familiar with linked lists and the usual ways to implement trees. A good reference for this is [1].

In one space dimension, the data structure we use to keep track of the grids is a tree structure. Each grid in the grid hierarchy corresponds to a node in the tree. When a fine grid is nested in a coarse grid, we say the corresponding node in the tree is an offspring of the parent node corresponding to the coarse grid. Two subgrids in the same coarse grid are said to be siblings. Fig. 3a shows a grid structure with one coarse grid, three fine grids at level 1, and three fine grids at level 2. The corresponding tree structure which details the relationships between the grids is shown in Fig. 3b. The only nonstandard links in this tree are indicated by the dashed lines. These additional pointers make the operation of finding all grids at a given level easy.

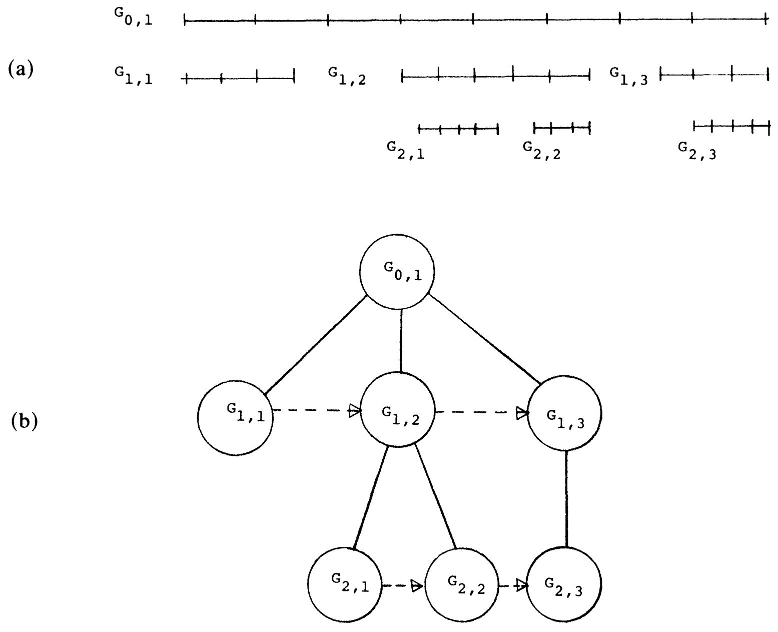


FIG. 3. *A one-dimensional grid structure and its associated tree data structure.*

To implement the tree, we would like each node to have a fixed number of items of information describing the grid. Since a grid can have an arbitrary number of subgrids, the tree is implemented by having each node point only to its first offspring, and the offspring points to its next sibling.

The information that is stored in a node for each one-dimensional grid is:

1) grid location,
2) Δx, Δt, t,
3) number of rows and columns,
4) parent grid pointer,
5) first subgrid pointer,
6) sibling pointer,
7) solution storage pointer.

Even in the unlikely event of a calculation having 50 fine grids, with 15 pieces of information per grid, this is only 750 words of storage. This is a very small amount of storage overhead compared to the amount of storage needed for the solution itself. By using locally uniform fine grids, we save the storage overhead found in irregular mesh refinement approaches, which is typically proportional to the number of grid points. (For a discussion of the CPU overhead in this approach to adaptive mesh refinement see [3].)

We emphasize two things about the tree data structure. Most integration algorithms (in particular, in [3]) have an information flow which follows path links in the tree, so there is no processing time overhead associated with them. For example, the fine grid typically gets boundary values from an underlying coarse grid. This information is directly available from the fine grid node, without having to traverse the entire tree. The second point about the data structure is that it needs to be dynamic in an unusual way for trees. Fine grids will be created and/or removed as needed during a calculation. This will occur more frequently for the finest level grids than the coarser grids. The data structure is changed by creating a new bottom half of the tree and joining it with the old top half of the tree. The nodes from the old bottom half are added to a list of free nodes.

There are several complications in the two-dimensional version of a grid structure that lead us to generalize the tree structure. First is the possibility of more than one coarse grid, and thus, more than one root node of the tree. Secondly, a grid may be nested in more than one coarse grid and thus have several parents in the tree. The parent slot can be replaced by a pointer to a short linked list of parent grids. Third, in two dimensions, grids at the same level of refinement may overlap, and so we add a pointer to a list of intersecting grids to the information which is stored for each node. In addition, for rotated grids we also store $\sin(\theta)$, $\cos(\theta)$, where θ is the angle of rotation of the grid. Fig. 4a shows a sample two dimensional grid structure, with two grids at each of the three levels; Fig. 4b shows the corresponding data structure. Conceptually, it can still be thought of as a tree.

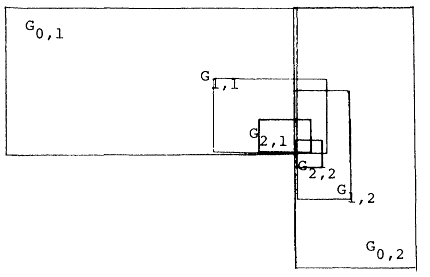
At this point we discuss some of the overhead associated with the use of rotated rectangular subgrids. In any operation between grids that are rotated with respect to each other, each indexing operation will involve an extra calculation. Suppose we need the coordinates of point (i, j), for example to interpolate an initial solution value for a point in a newly created fine grid. If the grid is not rotated, the typical calculation looks like

$$\begin{pmatrix} x_{ij} \\ y_{ij} \end{pmatrix} = \begin{pmatrix} c_x + (i-1)\,\Delta x \\ c_y + (j-1)\,\Delta y \end{pmatrix},$$

where (c_x, c_y) are the coordinates of the lower left hand corner of the fine rectangle. In the rotated case, the calculation looks like

$$\begin{pmatrix} x_{ij} \\ y_{ij} \end{pmatrix} = \begin{pmatrix} c_x + \cos(\theta)(i-1)\,\Delta x - \sin(\theta)(j-1)\,\Delta y \\ c_y + \sin(\theta)(i-1)\,\Delta x + \cos(\theta)(j-1)\,\Delta y \end{pmatrix}.$$

(a)



(b)

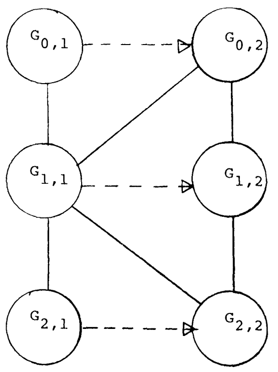


FIG. 4. *A two-dimensional grid structure and its associated graph.*

This amount of overhead is not bad. The mathematically difficult problem concerns the boundary conditions for a fine grid. In cases where the solution to a pde is discontinuous, the numerical method should be conservative. This rules out straightforward bilinear interpolation, and requires much more difficult interface equations (see [4] for a discussion of this problem). In such cases we use nonrotated grids, since the interface between grids then degenerates to a one-dimensional interface which is much more mathematically tractable. However, this is an active area of research today, and we do not believe that final conclusions can be drawn now.

The final data structure in our problem allocates storage for the solution on each grid from one long solution storage array. This array is managed by keeping a list of free chunks of storage, sorted by their location in the array. When a grid is created, its request for a certain number of words of storage is satisfied by taking contiguous storage from the first free block which is big enough. This is a first-fit algorithm. Such algorithms have been shown [13] to be preferable in most cases to best-fit algorithms, in which storage is allocated from the block that is closest in size to the requested amount.

3. Algorithms to generate the grid structure. In this section we present the algorithms used to generate the fine grids. We describe the algorithms in two dimensions, but they generalize immediately to higher dimensions. The algorithm starts with a rectangular coarse grid. Based on estimates of the error, grid points are flagged as needing to be in a grid with finer mesh width. The problem is this: given a list of flagged grid points, how should (rotated) rectangular subgrids be created to minimize the total area of the refined grids, so that each flagged point is interior to a fine grid unless it is on the boundary of the physical domain. Since the work of integrating the solution on a fine grid is proportional to its area, it is clear that we would like to minimize the area of the coarse grid which is unnecessarily refined.

This is a difficult problem, and it is difficult to find a foolproof algorithm which works in all cases. Our approach is to use a simple grid generation first and then evaluate the resulting grids to try to detect when it fails. We then use a safer but more expensive grid generation algorithm. The complete procedure consists of:

1) a clustering algorithm to decide which flagged points go together in one fine grid;

2) a grid fitting algorithm, which fits a rotated rectangle to each cluster;

3) an evaluation step, which detects the failure of the simple clustering algorithm by measuring the area of the proposed fine grid which is unnecessarily refined. We would also like as few grids created as possible, since there is computational overhead associated with the integration of the boundary of each fine grid.

We describe the first pass through the clustering procedure and then the alternate algorithms used in the difficult cases. Using either clustering algorithm, the fitting of the rectangle is the same. In the nonrotated case, the new grid is defined by finding the dimensions of the rectangle enclosing the cluster. The procedure for finding the orientation for rotated subgrids is discussed at the end of this section. (For time dependent adaptive calculations, the grids are then enlarged to include a buffer zone of a few mesh widths, to lengthen the interval between regridding operations. How this buffer zone is added can affect the amount of overlap between the grids.)

The clustering algorithm serves two purposes. The first is to separate the flagged points which come from spatially separated phenomena (see Fig. 5a). This is simple to do using a nearest neighbor algorithm. Start with one flagged point in a cluster. Add flagged points if the distance between the point and the point nearest to it in the cluster is small enough, typically two mesh widths or less. Since the flagged points come from a regularly ordered grid, this algorithm runs in time approximately linear in the number of flagged points n, rather than the worst case $O(n^2)$.

The second purpose of the clustering algorithm is to break up one nearest neighbor cluster if it leads to an inefficient grid. In Fig. 5b, if the entire cluster were fit with one grid (the dotted rectangle) an unacceptably large area would be refined. Instead, the cluster should be divided in two. Step 3, the evaluation step, would detect this using the simple approximation of taking the ratio of the number of flagged grid points to the total number of coarse grid points in the new fine grid. If the ratio falls below a cutoff, typically between $\frac{1}{2}$ and $\frac{3}{4}$, the points must be reclustered.

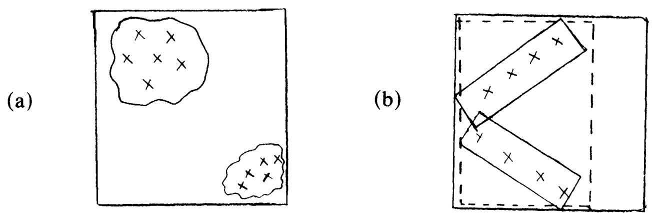


FIG. 5. *The clustering algorithms divide up the flagged grid points that should go together to make a new fine grid.*

The nearest neighbor clustering works well when entire regions of the domain need to be refined, and each such region is separated from the next. This occurs in transonic flow, for example, where the leading and trailing edges of the airfoil typically need refinement. If nearest neighbor clustering does not work, the assumption is that the points needing refinement lie along either a long curved front (as in Fig. 2), or several intersecting fronts (as Fig. 1 indicates). Since there may be a lot of scatter in the flagged points, it is difficult to sense their direction or distribution. For this reason, we next use data structures to organize the flagged points to understand how they are related, so that a smart subdivision of the points can be made.

We first connect the flagged points into a minimal spanning tree (MST). A MST is the connected acyclic graph connecting all the points so that the sum of the lengths of the edges is a minimum [1]. In this graph, each flagged point is connected to its nearest neighbor. The hard part now is deciding how to break the graph into different subclusters. An iterative algorithm immediately comes to mind. Start with a cluster of one point at the end of the MST. Add neighboring flagged points into the cluster if the resulting grid is still acceptable. Since the grid fitting part of the algorithm is easy (discussed next), this algorithm is feasible even though it is iterative. Fig. 6 shows a sample set of points, their MST and the three rectangles this iterative algorithm produces, starting at the left.

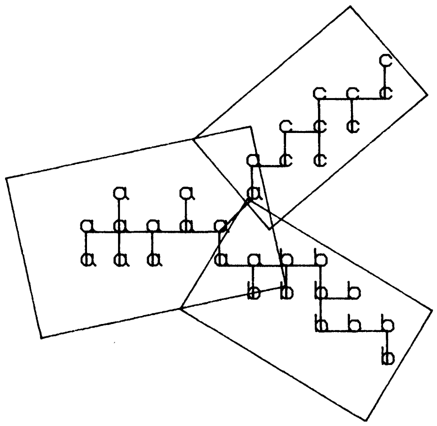


FIG. 6. *Three subgrids are generated from the* MST *connecting the flagged points.*

In practise, there are two changes made in the above algorithm. First it is less expensive to start with several points in a cluster rather than one. We take the original cluster and (inefficient) rectangle formed by the nearest neighbor algorithm, and repeatedly bisect it in the long direction until the component grids are acceptably efficient. The components are then put back together (if possible) in the iterative merging step based on the MST. Fig. 7a illustrates a "worst case" example which still works well. The flagged points make a cross pattern. Bisection makes the least obvious cuts, yielding 8 clusters. However, in Fig. 7b the merging step puts them back together nicely.

The second change we make in the algorithm as stated comes from the fact that the MST is not unique. Figs. 8a and 8b give two MST's for the same point set. The problem in Fig. 8 is the long path length between neighboring points. Since our flagged points come from a regular grid structure, this problem can occur frequently. We therefore generalize the MST by connecting a point to All its Nearest Neighbors, to make an ANN graph.

In constructing the MST or ANN connecting clusters of points, a little care must be taken in the definition of the location of a cluster of points. For example, in Fig. 9 the mean of the points would suggest that rectangles A and B be connected, but

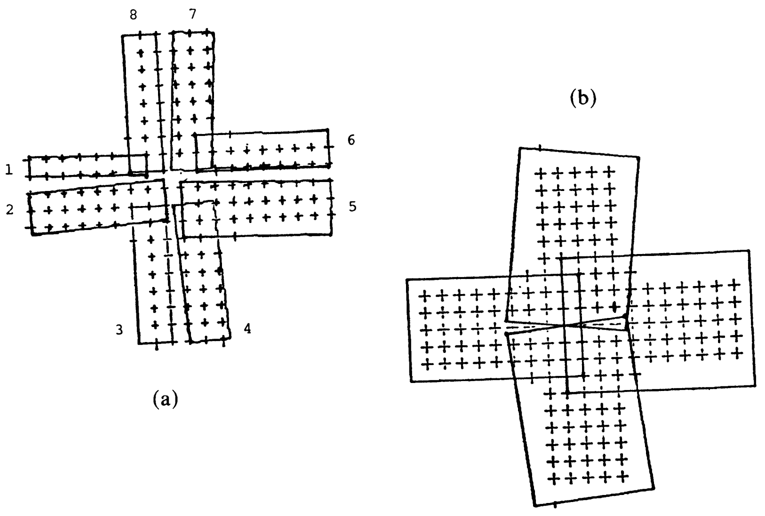


FIG. 7. *Bisection of the originally proposed rectangle yields* 8 *clusters. The merge step produces* 4 *fine grids.*

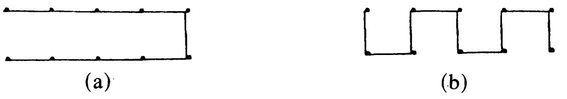


FIG. 8. *Two different* MST's *for the same point set.*

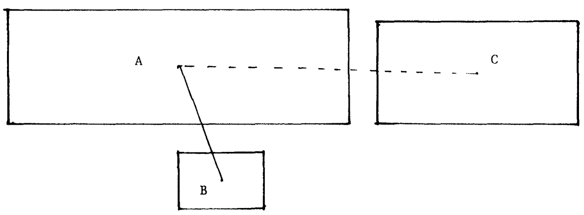


FIG. 9. *Rectangles A and C should merge, even though rectangles A and B are closer.*

clearly it is better to merge A and C. Other measures, such as corner separation, must be used. There are other graphs that provide useful structures with which to think about these problems. For example, clusters AB and AC in Fig. 9 would both be connected in a Relative Neighbor Graph [17], where two clusters are connected if no other cluster is closer to them both. We do not actually construct an RNG however, and admittedly, only approximate MST and ANN graphs are really needed in this work.

Returning to the grid generation algorithm, we now discuss how to generate a rotated rectangle to enclose all flagged points in one cluster. In addition, if flagged points have an orientation, the subgrid should have that orientation too. Let x_i, y_i, $1 \leqq i \leqq n$ be the coordinates of the flagged points, and $\bar{x}$, $\bar{y}$ their mean. Consider the symmetric matrix

$$M^tM = \begin{pmatrix} \sum_i x_i^2 - \bar{x}^2 & \sum_i x_i y_i - \overline{xy} \\ \sum_i x_i y_i - \overline{xy} & \sum_i y_i^2 - \bar{y}^2 \end{pmatrix},$$

where

$$M=\begin{pmatrix} | & | \\ x_i-\bar{x} & y_i-\bar{y} \\ | & | \end{pmatrix}.$$

This matrix determines an ellipse with the same first and second moments as the flagged points [9]. The axes of the ellipse are the eigenvectors of the 2 by 2 matrix M^tM, which we use to determine the orientation of the rectangle. The most expensive parts of this algorithm is the determination of the dimensions of the rectangle, given its orientation, so that all flagged points are included.

It turns out that this algorithm is related to a total least squares fit [11] of the data points in the following way. Suppose we look for a linear fit to the flagged points by a line through the mean,

$$(y-\bar{y})=m(x-\bar{x}),$$

so that we have

$$\left\{\begin{pmatrix} x_1-\bar{x} \\ x_2-\bar{x} \\ \vdots \\ x_n-\bar{x} \end{pmatrix}+\begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_n \end{pmatrix}\right\} m=\begin{pmatrix} y_1-\bar{y} \\ y_2-\bar{y} \\ \vdots \\ y_n-\bar{y} \end{pmatrix}+\begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_n \end{pmatrix}. \tag{3.1}$$

The slope of the line m is determined so that the Frobenius norm of the perturbation vectors $e=(e_1, e_2, \cdots, e_n)^t$ and $r=(r_1, r_2, \cdots, r_n)^t$ is a minimum. Rewrite (3.1) as

$$([x-\bar{x}|y-\bar{y}]+[e|r])\begin{bmatrix} m \\ -1 \end{bmatrix}=0.$$

For the smallest perturbation, take the perturbation matrix $C=[e|r]$ to be the smallest singular value of the matrix $[x-\bar{x}|y-\bar{y}]$. This matrix is just the matrix M of the flagged points. The solution vector $(m, -1)^t$ is the singular vector corresponding to the smallest singular value. This singular vector of M is one of the eigenvectors of the matrix M^tM above, and so both derivations give the same rectangle orientation. By using the total least squares derivation, we see that we are finding the slope of the line through the mean of the points which minimizes the sum of the squares of the distances from each point to the line.

One of the goals in grid generation is to create subgrids with total area as small as possible. In two dimensions it is computationally feasible to construct the minimum area rectangle enclosing a set of points. We will briefly describe how to do this and compare it with our procedure above. To construct a spanning rectangle, first find the convex hull of the set of points. In two dimensions, this may be done in $O(n \log n)$ operations, where n is the number of flagged points to be enclosed [12]. The next step makes use of a theorem by Freeman and Shapira [10] proving that the minimum area rectangle has a side collinear with the convex hull. It remains to check each side of the hull for the rectangle with the minimum area. For each line segment on the hull, the area of the spanning rectangle may be determined in time $O(h)$, where h is the number of vertices on the boundary of the convex hull, by carefully figuring which vertices on the boundary anchor the rectangle. Finally, choose the rectangle with the minimum area.

This algorithm does not generalize easily to higher dimensions, where the convex hull takes $O(n^2)$ operations to compute. Even in two dimensions, it is much more expensive than using the ellipse method. Since the minimum spanning rectangle depends only on the convex hull of the set of points, it will be aligned with the points only if the clustering algorithm aligns the points first. In experiments comparing the minimum area spanning rectangle with moment generated rectangles, the latter typically has only 5 to 10% larger area than the former.

This performance of the ellipse algorithm has been heuristically explained in [2] in the following way. Suppose the sides of the rectangle are oriented in the directions of the orthonormal unit vectors r_1, r_2. The length of the side parallel to r_1 is $\max Ar_1 - \min Ar_1$, where A is the n by 2 matrix of the flagged points,

$$A=\begin{pmatrix} x_1 & y_1 \\ \vdots & \vdots \\ x_n & y_n \end{pmatrix}$$

and similarly for r_2. The max and min are taken componentwise over the vector. If the points are clustered so that there are no extreme outlying points, (which we certainly expect the clustering algorithm to accomplish), the length of a side is

$$\max Ar_1 - \min r_1 \approx 2\|Ar_1\|_\infty.$$

Thus we consider the maximum norm problem of minimizing the area of the enclosing rectangle,

$$\min_{\substack{r_1,r_2\\ r_ir_j=\delta_{ij}}} \|Ar_1\|_\infty \|Ar_2\|_\infty. \tag{3.2}$$

If (3.2) is approximated using the 2-norm, this problem can be easily solved using the fact that

$$\|Ar_1\|_2^2\|Ar_2\|_2^2 = r_1^tA^tAr_1r_2^tA^tAr_2,$$

but for the orthonormal vectors r_1r_2,

$$r_1^tA^tAr_1 + r_2^tA^tAr_2 = \text{constant}.$$

The problem again becomes that of minimizing $\|Ar\|_2$ over unit vectors r, giving the same orientation vectors that the moment generated ellipse gives. The area of the enclosing rectangle is related to the area of the minimum enclosing rectangle as the 2-norm is related to the maximum norm. In general there can be a large difference between those two norms. However, if we use a smart clustering algorithm, this will not be the case.

We end with examples of the grid generation package on several point sets which illustrate the effect of the efficiency parameter in the final step. In Fig. 10a, approximately 85% of the points in each of the six fine grids are flagged. In Fig. 10b only 45% were required to be flagged, and so larger grids were created. Figs. 11a and 11b illustrate a similar phenomenon on a different set of points.

There is still much more that could be done to come up with a fast, reliable grid generation package. For example, in Fig. 11a, the higher-efficiency-rated grids do not have a smaller total area refined, since the fine grids overlap more than in Fig. 11b. Other measures for evaluating grids might thus be beneficial. There are also alternative grid generation algorithms which seem reasonable but have not been tested. For example, based on the MST one could use the diameter of the graph to indicate the layout of the points. In problems with a discontinuous solution, if we assume that the flagged points follow the front, the diameter should approximate the shape of the front. Clusters can now be formed by segmenting the diameter. This is similar to an

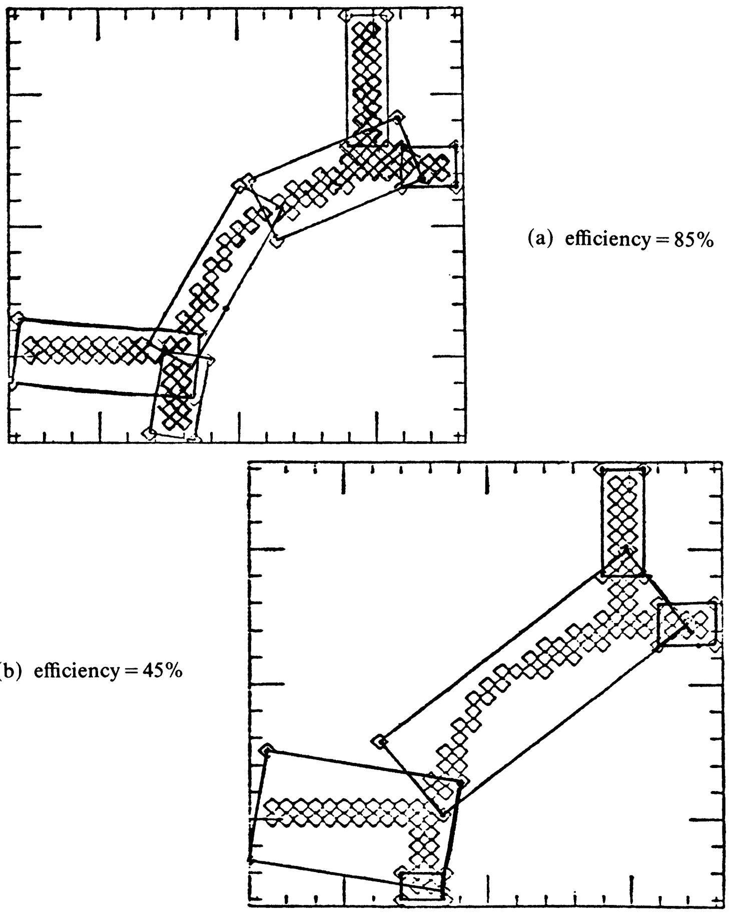


FIG. 10. (a) 85% *of the coarse grid points in the fine grid are flagged*; (b) *only* 45% *are flagged.*

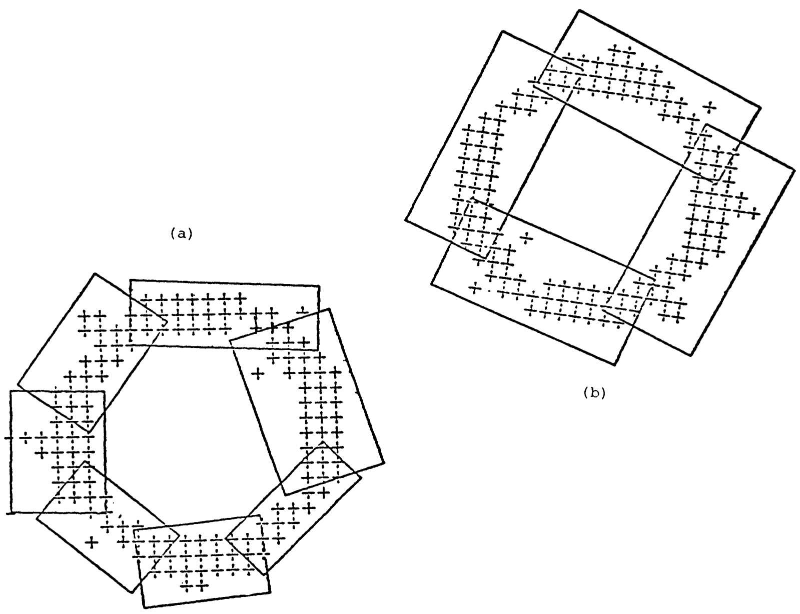


FIG. 11. (a) 75% *of the coarse grid points in the fine grids are flagged*; (b) *only* 50% *are flagged.*

algorithm in [14] which was used to recognize curved objects in the plane. Although the algorithms presented here are still experimental and under development, they have already been incorporated into adaptive mesh refinement programs for the solution of pdes. Although not optimal, they have been proven successful and efficient in the automatic generation of adaptive subgrids.

Acknowledgments. I would like to thank Doug Baxter for his participation in the development of the grid generation algorithms. I also thank William Gropp for his participation in the early stages of the development of the mesh refinement program.

REFERENCES

[1] A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[2] D. Baxter, M.S. thesis, Dept. Computer Science, Stanford Univ., Stanford, CA, 1981.

[3] M. J. Berger and J. Oliger, *Adaptive mesh refinement for hyperbolic partial differential equations*, J. Comp. Phys., 53 (1984), pp. 484-512.

[4] M. J. Berger, *On conservation at grid interfaces*, ICASE Report No. 84-43, NASA, Langley, VA, September, 1984.

[5] M. J. Berger and A. Jameson, *Automatic adaptive grid refinement for the Euler equations*, AIAA J., 23 (1985), pp. 561-568.

[6] ——— *An adaptive multigrid method for the Euler equations*, in Lecture Notes in Physics 218, Soubbaramayer and J. P. Boujot, eds., Springer-Verlag, Berlin, 1985.

[7] J. Bolstad, Ph.D. thesis, Dept. Computer Science, Stanford Univ., Stanford, CA, 1982.

[8] S. Caruso, Ph.D. thesis, Dept. Mechanical Engineering, Stanford Univ., Stanford, CA, in preparation.

[9] H. Cramer, *Mathematical Methods of Statistics*, Princeton Univ. Press, Princeton, NJ, 1951.

[10] H. Freeman and R. Shapira, *Determining the minimum-area encasing rectangle for an arbitrary closed curve*, Comm. ACM, 18 (1975), pp. 409-413.

[11] G. H. Golub and C. van Loan, *An analysis of the total least squares problems*, SIAM J. Numer. Anal., 17 (1980), pp. 883-893.

[12] R. Graham, *An efficient algorithm for determining the convex hull of a finite planar set*, Inform. Proc. Letters, 1 (1972), pp. 132-133.

[13] D. Knuth, *The Art of Computer Programming*, Vol. 1, Addison-Wesley, Reading, MA, 1973.

[14] R. Nevatia and T. Binford, *Description and recognition of curved objects*, Artif. Intell., 8 (1977), pp. 77-98.

[15] J. Saltzman and J. Brackbill, *Applications and generalizations of variational methods for generating adaptive meshes*, in Numerical Grid Generation, J. Thompson, ed., North-Holland, Amsterdam, 1982.

[16] K. Stuben and U. Trottenberg, *Multigrid method: fundamental algorithms, model problem analysis and applications*, in Multigrid Methods, W. Hackbusch and U. Trottenberg, eds., Springer-Verlag, Berlin, 1982.

[17] G. Toussaint, *The relative neighborhood graph of a finite planar set*, Pattern Recognition, 12 (1980), pp. 261-268.