Generating a good quality mesh for modeling diffusion

E. W. Bowker[1], L. P. Gewali[2] and D. W. Pepper[3]
University of Nevada, Las Vegas
[1]*lincoln@cs.unlv.edu*
[2]*laxmi@cs.unlv.edu*
[3]*pepperu@nye.nscee.edu*

Acknowledgement: Research supported in part by a grant from DOE, NVOO.

Abstract

We consider the implementation issues of quadrilateral mesh generation algorithms and present their use for solving dispersion problems in environmental applications.

1 Introduction

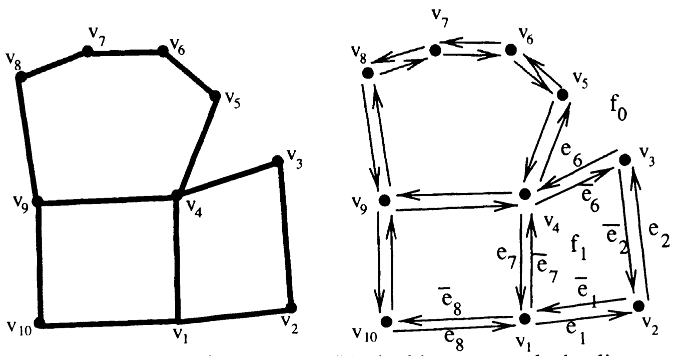
Generating a good quality mesh is a critical step in finite element applications. One approach for generating a quadrilateral mesh is to first generate a triangular mesh and then convert it to a quadrilateral mesh by combining adjacent triangles. Although this approach is simple to understand and easy to implement, the resulting quadrilateral mesh tends to contain trapped triangles. In this paper, we address the implementation issues of generating quadrilateral mesh directly. The two issues we consider are (i) the quality of a generated mesh, and (ii) the execution time of mesh generating algorithms. We use doubly connected edge list (DCEL) data structure to represent the mesh, which allows us to manipulate nodes, edges, and faces of the mesh quickly and elegantly. Our system is coded in JAVA/C++ which can be easily adopted to prepare it as a Java applet, which in turn can be executed from a computer with a web browser and connected to the Internet.

We also present the use of a generated orthogonal mesh for solving dispersion problems in environmental applications. In particular, we present the generation of isothermal lines using the example of heat conduction in two dimensions.

2 Data structure for representing mesh

The appropriate choice of data structure is very crucial for implementing mesh-generating algorithms. A two-dimensional mesh can be viewed as a planar straight line graph (PSLG). Two types of data structures have been proposed to represent a PSLG in computational geometry literature [1]. These are: (i) doubly connected edge list (DCEL) structure and (ii) quad edge structure. Although the latter is general enough to capture both the primal and the dual structure of a PSLG, we found the former to be simpler and more convenient for implementation. In DCEL representation, we store the mesh in three arrays: arrays of nodes, faces, and half edges. This is illustrated in Figure 1. Figure 1a is a quadrilateral mesh consisting of three **mesh elements** or **faces** (f1, f2, f3), ten **nodes** ($v_1, v_2, ..., v_{10}$) and twelve line segments ($s_1, s_2, ...s_{12}$). Each line segment s_i is represented by a pair of half edges e_i and $\overline{e_i}$. Half edges e_i and $\overline{e_i}$ are **twins** of each other and are directed in opposite directions. The direction of half edges is chosen to facilitate the traversal of the outer boundary of the polygon in the counterclockwise direction. In order to facilitate the traversal of mesh elements quickly and conveniently, each of the half-edge, node, and face records are enhanced to contain additional adjacency information. The record for a half edge includes the following information.

- **Inc-face**(e_i): the face to the right of e_i.
- **Twin**(e_i): the twin edge of e_i.
- **Succ**(e_i): the successor edge of e_i in Inc-face(e_i).
- **Pred**(e_i): the predecessor edge of e_i in Inc-face(e_i).
- **Orig**(e_i): the node from which e_i originates.



(a): A planar graph (b): doubly connected edge list

Figure 1: Illustrating doubly connected edge list

The record for a face f_i includes:

- **bound**(f_i): one of the half edges bounding the face f_i.

The record for a vertex v_i includes:

- **coord**(v_i): the x, y coordinates of v_i.
- **Inc-Edge**(v_i): A half edge whose origin vertex is v_i.

The three arrays for the DCEL of Figure 1b are tabulated below.

Half-edge array

Half-edge	Origin	Twin	Pred	Succ	Inc-face
e_1	v_1	$\overline{e_1}$	e_8	e_2	f_0
$\overline{e_1}$	v_2	e_1	$\overline{e_2}$	$\overline{e_7}$	f_1
...	...	...	...	...	...
...	...	...	...	...	...
...	...	...	...	...	...

Node and Face arrays

Node	Coord(v_i)	Inc-Edge(v_i)	Face	Bound(f_i)
v_1	x_1, y_1	e_1 or $\overline{e_8}$	f_0	e_1
v_2	x_2, y_2	e_2 or $\overline{e_1}$	f_1	$\overline{e_6}$
...	...	...	f_2	$\overline{e_4}$
...	...	...	...	...

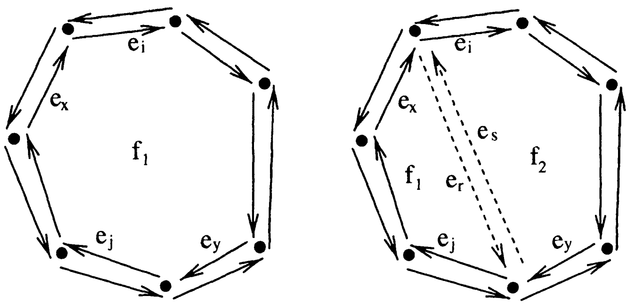


Figure 2: Illustrating face splitting

Each of the half-edge/face/node records may contain additional information warranted by a specific application. In fact, in our application we include additional information in each record. The following are some operations we have implemented for the DCEL data structure.

- **Traverse-face** (f_i): Traverse the half edge bounding a face f_i. With DCEL representation, we can traverse the boundary of f_i by following the successors of a half edge bounding f_i. This is done in $O(size(f_i))$ time, where $size(f_i)$ is the number of edges bounding face f_i.

- **Split-face**(f_1, e_i, e_j): Partition the face f_1 into two parts f_1 and f_2 by an edge connecting the origin vertices of half edges e_i and e_j as shown in Figure 2. Because of DCEL representation, splitting of a face can be done very quickly by creating two half edges e_r and e_s and by setting appropriate fields in the records of e_i, e_j, e_x and e_y.

- **Split-edge**(e_i, x, y): Split a half edge e_i and its twin edge $\overline{e_i}$ at an interior point (x, y). This is equivalent to replacing two half edges with four half edges and by adding a new node with coordinate (x, y) in the data structure. Appropriate fields of new half edges and their successors/predecessors are adjusted. All this adjustment takes constant time.

- **Remove-edge**(e_i): Remove the half edge e_i and its twin half edge $\overline{e_i}$. This might result in the combination of two faces into one if e_i and $\overline{e_i}$ have different incident faces. This adjustment takes $O(size(f_1) + size(f_2))$ time, where f_1 and f_2 are the incident faces of half edges e_i and $\overline{e_i}$, respectively.

3 Quadrilateral mesh generation

In this section we give a brief overview of approaches used for generating a quadrilateral mesh when the input domain is available in the form of a simple polygon. We also suggest a method for refining a convex face by using generalized splitters. While a polygon can always be converted to a triangular mesh using only the original vertices, this does not hold true for generating a quadrilateral mesh. For some polygons, we must introduce Steiner nodes to convert them to quadrilateral meshes. Since a number of triangulation algorithms are available, some researchers have suggested starting with a triangulation to generate a quadrilateral mesh. A triangular mesh can be converted to a quadrilateral mesh by combining adjacent triangles. However, this process suffers from the drawback that the resulting quadrilateral mesh is prone to contain trapped triangles [2].

One of the input parameters used for generating a quadrilateral mesh is the size $a \times a$ of the desired element, where a denotes the desired approximate length of the side of a quadrilateral mesh element. The requirement of approximate mesh element size forces us to introduce Steiner vertices both on the boundary and on the interior of the polygon. It is well known that the number of vertices on the boundary of a polygon must be even in order to make all mesh elements quadrilateral [2]. It is generally assumed that the desired size of the mesh element a is much smaller than the smallest rectilinear box that encloses the polygon. It is also fairly reasonable to assume that the size a of the mesh element is smaller than the smallest length boundary edge of the polygon.

We split the bounding edges of the polygon by introducing Steiner vertices so that the total number of vertices on the boundary is even. This is achieved by using the following strategy. We first identify the **longest** boundary edge of the polygon. We refer to other edges as **ordinary edges**. Each ordinary edge is partitioned into sub-edges of equal length so that the length of a sub-edge is as close to element size a as possible. It may be noted that the lengths of the sub-edges belonging to different edges need not be equal. If the total number of sub-edges in all ordinary edges is even then the longest edge is partitioned into an even number of sub-edges, each of length as close to a as possible. On the other hand, if the total number of sub-edges on all ordinary edges is odd then we partition the longest edge into an odd number of sub-edges. This check guarantees that the total number of sub-edges on the boundary is even.

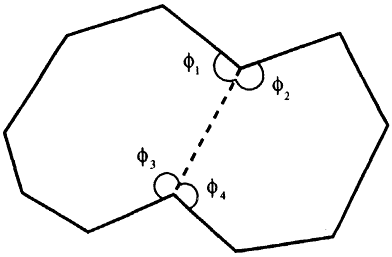


Figure 3: Illustrating angle criterion

Perhaps the most critical step in mesh generation is the partitioning of a face to sub-faces. Quadrilateral mesh elements are formed by successive partitioning of faces to make each sub-face a quadrilateral. The proper choice of a partitioning diagonal is very crucial. Various strategies have been suggested in recent years to pick a partitioning diagonal to increase the quality of the generated mesh [4]. A partitioning diagonal should make the smallest angle subtended by it to any incident edge as large as possible. This is illustrated in Figure 3, where we want to maximize the smallest angle among ϕ_1, ϕ_2, ϕ_3, and ϕ_4. This criterion is referred to as the **angle** criterion. It is desirable that the partitioning diagonal be of small length so that the error introduced by the newly inserted diagonal is minimized. We term this criterion as the **length** criterion. Also the length of the diagonal is desired to be close to a multiple of mesh element size a. This criterion is referred to as the **factor** criterion. Another criterion that has been suggested is the **symmetry** of partitioning. If the component faces induced by a partitioning diagonal are symmetrical the resulting mesh tends to have better quality. However, measuring the degree of symmetry is an intricate problem in itself. One easy way to impose symmetry is to make the area of component faces approximately equal to each other. Often, the criteria as described above can not be satisfied simultaneously. One way is to devise an objective **cost function** by formulating a linear combination of several criteria. For example, we could devise a cost function as follows.

$$Cost = c_1\phi + c_2 l + c_3\sigma \qquad (1)$$

where c_1, c_2, and c_3 are weighting factors and ϕ, ℓ, and σ are the quantities measuring the angle, length, and factor criteria.

The approach based on partitioning a face by a suitably chosen diagonal does not guarantee a good quality mesh. In fact, even for convex polygons, the diagonal induced splitting need not produce

Transactions on Ecology and the Environment vol 25, © 1998 WIT Press, www.witpress.com, ISSN 1743-3541

a good quality mesh. Our preliminary investigation shows that for convex faces, it is sometimes desirable to split it not by a diagonal but by generalized $X-Y$ structure and its variations. We are investigating this issue and the result will be reported in the future.

In our proposed algorithm, we first split the boundary edges into sub-edges of size close to the desired mesh element size a. We then decompose the polygon into convex components by resolving reflex vertices. The reflex vertices are resolved by repeatedly inserting diagonals between a reflex vertex and a reflex or non-reflex vertex. In doing so, we check only two criteria: angle and length. The resulting convex pieces are further partitioned into smaller size (4, 6, 8, or 10) convex faces by using diagonal and $X-Y$-structured splitters. The smaller-sized convex faces can be converted to quadrilaterals by using template-matching techniques.

4 Computing dispersion

We now consider the application of the generated mesh to compute dispersion in a two dimensional domain modeled by an orthogonal polygon. The mesh elements are isothetic rectangles. The specific dispersion problem we consider is the computation of heat conduction in two dimensions. The boundary edges of the polygon are categorized into three types: (a) **Dirichlet**, (b) **Neumann**, and (c) **convective**. The initial values of temperature, thermal conductivity, etc., associated with each boundary node can be interactively entered by the user. For interior nodes, the initial values of temperature are taken to be zero. The user can also enter the level of mesh refinement interactively. The whole program is written in JAVA/C++. The input polygon can either be entered interactively by clicking mouse buttons or by reading from a previously stored file. Figure 5a shows an orthogonal mesh created by the code.

The governing equations for the diffusion of heat can be written as [5]

$$\nabla \cdot (k \nabla T) + Q = 0 \tag{2}$$

where

$$k \nabla T + q + h(T - T_{\infty}) = 0 \tag{3}$$

defines the generalized boundary condition applied along boundary Γ. Thermal conductivity, defined by k, can either be constant or non-uniform according to material properties; h is the convection coefficient, T_∞ is the ambient temperature, q is flux, and Q denotes source/sink.

The weak formulation obtained by applying the Galerkin Method of Weighted Residuals for quadrilateral elements can be written in matrix form as [5, 6]:

$$[K]\{T\} = \{f\} \tag{4}$$

where $[K]$ is the stiffness matrix obtained from the inner product. For example,

$$T(x,y) = \Sigma N_i(x,y)T_i \tag{5}$$

$$[K] = \int_\Omega [\nabla N_i(k\nabla N_j)]d\Omega \tag{6}$$

$$\{f\} = \int_\Gamma Nqd\Gamma + \int_\Omega NQd\Omega \tag{7}$$

where Ω is the two dimensional problem domain and Γ denotes the boundary where q is applied.

The element equations are assembled to form a global stiffness matrix. The matrix equation for temperature is solved using Cholesky (Skyline) method [5]. The newly generated temperature values are used as initial temperatures and the updated matrix equation is solved again to obtain new temperature values. This process is iterated until the difference between successive temperature values is less than a prescribed tolerance number ($\epsilon \leq 10^{-4}$).

4.1 Generating isothermal lines

In order to visualize the generated solution, we have developed and implemented an algorithm for plotting isothermal lines by using temperature values at nodal points. Let $T \equiv t_1 < t_2 < t_3 < ... < t_k$ be the ordered temperature list for which we want to generate isothermal lines. Our approach is based on first interpolating the intersection of contour lines with each edge of the mesh elements. Each mesh element is processed in turn to generate isothermal lines. The interpolation of the intersection points of isothermal lines with element edges is illustrated in Figure 4a.

Let t_a and t_b be the temperature values at nodes a and b on an edge (a,b). By comparing t_a and t_b with entries in the ordered list T, we find the range of temperature values $t_i < t_{i+1} < ... < t_j$ for the interior of edge (a, b). The exact location of points corresponding to $t_i, t_{i+1}, ...t_j$ on edge (a, b) is determined by linear interpolation of range $t_a \sim t_b$ on the length of edge (a, b). This interpolation is repeated on all edges. After finding intersection points on all edges the next step is to connect them appropriately. The connection procedure on a mesh element is shown in Figure 4b. Our assumption for using linear interpolation is stated below.

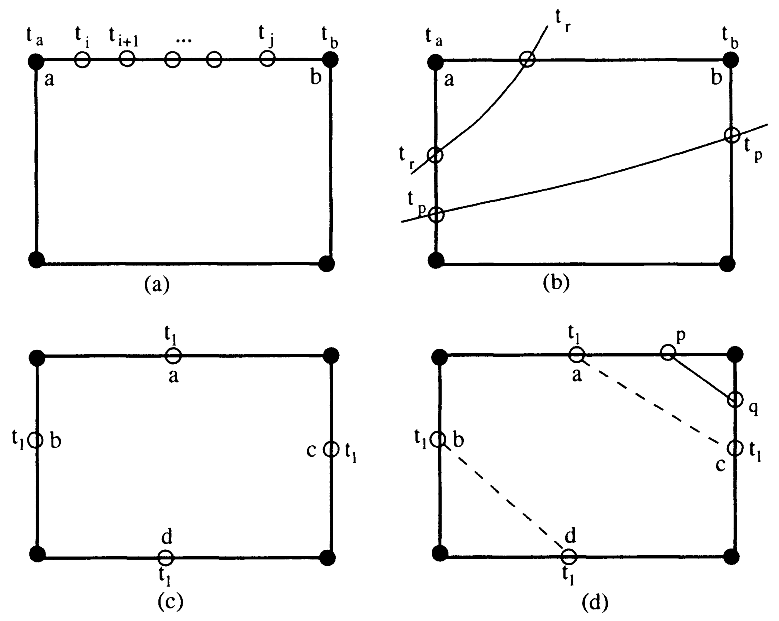


Figure 4: Illustrating interpolation and ambiguity

Assumption 1: If an isothermal line intersects a mesh element then it intersects with its boundary at no more that two points.

It may be noted that to make Assumption 1 acceptable in practice, we need to make the mesh element size much smaller than the size of the polygon. Isothermal lines inside an element are generated by connecting interpolated points corresponding to the same temperature values by a line segment. An ambiguity arises when a mesh element contains four points with the same temperature as shown in Figure 4c: it is not clear which pair to connect - (a, b) and (c, d) or (a, c) and (b, d). We can resolve a four-point ambiguity if another pair have already been connected as illustrated in Figure 4d, where the connection between p and q suggests that a should be connected to c and b should be connected to d. Repeating the point connection procedure for all mesh elements, the required isothermal lines for the whole mesh are obtained. It is easy to establish that the execution time of the above described isothermal line plotting algorithm is $O(m * k)$, where m is the number of mesh elements and k is the number of isothermal lines. Figure 5 shows snapshots of the execution of our implementation.

5 Discussion

We examined the implementation issues of quadrilateral mesh generating algorithms and presented the use of an orthogonal mesh for solving steady state diffusion problems. We are planning several enhancements to our implementation. One enhancement is the use of an unstructured mesh, rather than orthogonal, for dispersion computation. Work in this direction is in progress and the results will be reported in the future. For plotting isothermal lines, we have used line segment connections in each element. It may be appropriate to smooth out the isothermal lines by the use of B-splines [3] and we are currently investigating this issue. The next step is to extend the system to generate hexahedral elements in three dimensions.

References

[1] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O., *Computational Geometry*, Springer-Verlag, Berlin and Heidelberg, 1997.

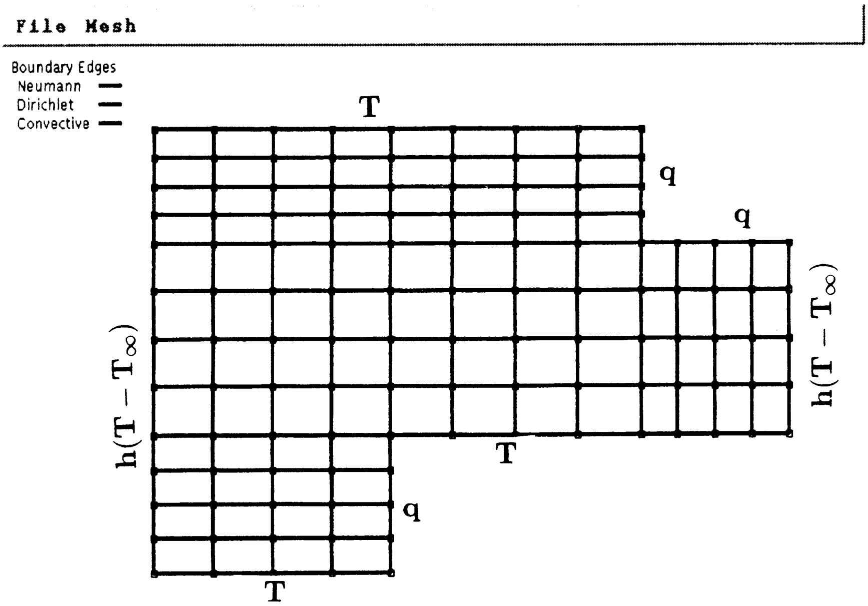
[2] Bose, P. and Toussaint, G., Characterizing and Efficiently Computing Quadrangulations of Planar Point Sets, to Appear in Computer Aided Design.

[3] Hearn, D. and Baker M,. *Computer Graphics*, Prentice Hall, 1986.

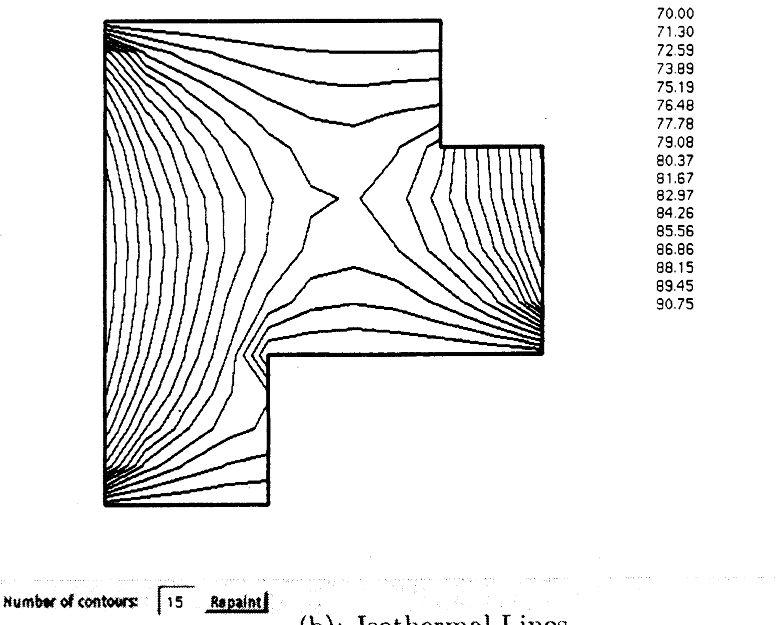
[4] Nowottny, D., Quadrilateral Mesh Generation via Geometrically Optimized Domain Decomposition, *Proc.of the 6th Meshing Roundtable*, Brigham Young University, pp. 309-320, 1997.

[5] Pepper, D. and Heinrich, J. C., *The Finite Element Method, Basic Concepts and Applications*, Hemisphere Publishing Co (Taylor and Francis), 1992.

[6] Zienkiewicz, O. C. and Taylor, C., *The Finite Element Method,* 4th Edition, McGraw Hill, London, UK, 1991.



(a): Generated Mesh



(b): Isothermal Lines

Figure 5: Snapshot of Execution