A Refinement Algorithm and Dynamic Data Structure for Finite Element Meshes[1]

Randolph E. Bank[2]

Andrew H. Sherman[3]

TR 159

[1] Work supported in part by NASA under grant NSG-1632 and by The Office of Naval Research under grant N00014-80-C-0645.

[2] Department of Mathematics, University of Texas at Austin, Austin, TX 78712. Currently on leave at Department of Computer Sciences, Yale University, New Haven, CT 06520.

[3] Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712.

1. Introduction

PLTMG [5] is a prototype finite element program designed to solve elliptic boundary value problems of the form

$$-\nabla\cdot(a\nabla u) + b\cdot\nabla u + cu = f \qquad \text{in } \Omega\subset R^2$$
$$u = g_1 \qquad \text{on } \partial\Omega_1 \qquad (1.1)$$
$$\frac{\partial u}{\partial n} = g_2 \qquad \text{on } \partial\Omega-\partial\Omega_1$$

where Ω is a general region of R^2. The program is based on a Galerkin procedure using C^0 piecewise-linear triangular finite elements and has several interesting features. First, PLTMG allows either user-controlled or automatic triangulation. The user must specify a crude initial triangulation of Ω. He can then either specify how this initial triangulation should be refined by assigning weights to the vertices or allow PLTMG to adaptively refine the triangulation. The adaptive procedure interleaves triangulation with generation and solution of the associated discrete problem. At each triangulation step it estimates the error in all current unrefined triangles and refines only those triangles in which the error is large. (Our error estimation procedure draws on the ideas of Babuska and Rheinboldt [1].)

Second, PLTMG uses a multi-level iterative method to solve the resulting linear equations [2-5]. This is an iterative procedure in which iterations are carried out on several triangulations of Ω of differing degrees of refinement. The abstract computational complexity of this algorithm is of optimal order; that is, $0(nv)$

operations are required to solve a problem with nv unknowns to a level of accuracy commensurate with the discretization error.

We will not discuss here the details of either of these features but rather refer the reader to [4, 5]. What is important to note is that implementation of either of these features on a practical level requires one to have the ability to generate, store, and modify several triangulations. Since the multi-level iterative method is extremely efficient, it cannot be assumed that the cost of solving linear systems dominates the costs of other operations such as mesh manipulation and discretization. For this reason, it is essential to have methods for generating, modifying and using triangulations which are as efficient as possible.

To meet these demands, we have developed a general mesh refinement procedure for triangular elements - general in the sense that it can accomodate an essentially arbitrary refinement rule specified through an external Fortran function. This allows us to use the same algorithm (indeed the same code) for both user-specified and adaptive grid generation by simply changing the function. We view the refinement procedure as generation of a "triangle tree" in which the nodes correspond to triangles, and branches extend downward from the root "triangle" Ω to the user-specified triangles, to their refinements, etc. The leaf nodes correspond to unrefined triangles in the finest mesh generated so far. Our data structures correspond closely to this interpretation of the refinement process. We have experimented with several data structures in various

versions of PLTMG over the past several years and feel that the one we describe here is the best we have yet found. Other approaches to the problem are described by Simpson [7].

PLTMG is designed to run as a self contained in-core program, and the data structure we describe thus reflects our view of an appropriate compromise between space and time. For a finest mesh containing nv vertices, the version of PLTMG using this data structure requires about 30 nv storage locations: 16 nv for sparse matrices, right hand sides, and the solution, 11 nv for the triangle-tree data structure, and 3 nv temporary work space. As a rule the algorithms we use for computing the necessary "standard" information about triangles on the tree or vertices in the grid run in constant time; that is, worse than table-look-up by at most a constant factor. Earlier versions of PLTMG used more table lookup and storage but ran somewhat faster (approximately 15%). At this writing, we have not found a way to significantly reduce the current storage requirements for the triangle-tree (short of "programming tricks" such as packing more than one integer to a word) without sacrificing at least some of our time-efficient algorithms.

The remainder of this paper is organized into four sections. In Section 2, we discuss mesh refinement in general and present the particular algorithm used in PLTMG. In Section 3 we define the triangle-tree data structure that we have devised, and in Section 4 we describe the accompanying algorithms for generating the standard information about triangles and vertices. Finally, in Section 5,

we describe how the refinement procedure can be extended to the case of C^0 piecewise-bilinear rectangular finite elements. In some respects, the rectangular refinement procedure is less complicated, and this is reflected in some minor simplifications which occur in the corresponding "rectangle tree" data structure. As in the case of triangles, the algorithms for computing the standard information run in constant time. A different refinement procedure for rectangles and a correspondingly different data structure are described by Rheinboldt and Mesztenyi [6].

2. Mesh Refinement

In the context of finite element methods, the basic idea of any mesh refinement algorithm is to generate a set of triangles that covers Ω and can be used to generate a discrete representation of the differential equation (1.1). Ideally, the total error in the solution to the discrete equation should meet some user-specified tolerance (as measured in an appropriate norm) and should be distributed equally over the triangles. This means that there will usually be a large number of triangles (to meet the tolerance) whose sizes vary depending in some way on the local smoothness of the solution to the continuous differential equation (to equi-distribute the error). It is unrealistic to expect a user to specify by hand the vertices of what may turn out to be thousands of triangles - hence the need for automatic mesh generation algorithms.

For use in PLTMG, automatically generated triangulations must satisfy two important properties:

(i) The size of each interior angle of each triangle must be bounded away from zero.

(ii) The transition from large to small triangles in the mesh must be "smooth". (That is, the size difference between adjacent triangles must be bounded.)

Both of these properties are motivated by finite element theory; they basically disallow long thin triangles. In addition, they simplify the overall triangulation process.

A number of schemes could be used to actually carry out the construction of a sufficiently-refined triangulation from a limited amount of user input data (see Simpson [7] for a survey). In PLTMG we have adopted an approach that allows for either user-controlled refinement or adaptive refinement. This is accomplished by viewing triangulation as a process in which one examines triangles in a mesh and occasionally asks: "Should this triangle be refined?" By allowing a logical-valued function to answer the question and simply changing the actual function supplied as an argument to the triangulation subprogram, either form of refinement can be accomodated. In PLTMG, we refer to the logical function as DVTEST.

We now describe the mesh refinement algorithm used in PLTMG, and for which our data structure was designed. For convenience we assume Ω is a polygon, although PLTMG makes provision for triangles on the boundary of Ω to have one curved boundary edge.

Initially, the user supplies a coarse triangulation $\mathcal{T}_0$ of Ω consisting of a small number of triangles t_i, $1 \le i \le nt_0$, that we call macro triangles (cf. Figure 2.1a). Each triangle t_i contains three vertices ν_i^j, $1 \le j \le 3$, and three edges ε_i^j, $1 \le j \le 3$, with ε_i^j opposite ν_i^j (cf. Figure 2.1b). It is convenient to assign global numbers to the vertices and edges in $\mathcal{T}_0$, denoted by v_k, $1 \le k \le nv_0$,

and e_k, $1 \leq k \leq ne_0$, respectively. Thus for $1 \leq i \leq nt_0$ and $1 \leq j \leq 3$, $\nu_i^j = v_k$ for some k, $1 \leq k \leq nv_0$, and $\varepsilon_i^j = e_\ell$ for some ℓ, $1 \leq \ell \leq ne_0$. Throughout this paper, we will view local designations (e.g. ν_i^j) and global designations (e.g. v_k) as interchangeable names for a unique entity, and we will use whichever designation makes more sense in context.

To ensure that condition (i) above is met, we allow only two types of triangle subdivision: regular and "green". In regular subdivision (cf. Figure 2.2a) a triangle t_i is divided into four smaller triangles, denoted t_{s_i+j}, $0 \leq j \leq 3$, by joining the midpoints of its edges. (Here s_i is one larger than the highest-numbered triangle currently in the mesh.) Each of the four new triangles (called "sons of t_i") is geometrically similar to t_i (its "father"), so that regular subdivision never reduces the size of the interior angles.

In green subdivision a triangle t_i is divided into two smaller "green triangles", denoted t_{s_i} and t_{s_i+1} (s_i as above), by inserting a "green edge" joining a vertex ν_i^j to the midpoint of the opposite edge ε_i^j (cf. Figure 2.2b). Green subdivision may reduce the size of the smallest interior angle, so repeated use could violate (i). Hence we only use it to "clean up" the grid by removing degenerate quadrilaterals which remain after all regular subdivision has been completed. In Figure 2.2b, if it should later become desirable to refine t_i further (in adaptive refinement), the two green triangles are ignored and regular refinement is applied directly to t_i.

Each edge of a triangle t_i either is a boundary edge (of Ω) or is part of the perimeter of one or more other triangles in the grid. If t_i contains a boundary edge, it is called a boundary triangle. We define the neighbor of t_i across edge ε_i^j, denoted τ_i^j,* as the smallest regular triangle with one edge which completely overlaps ε_i^j. If ε_i^j is a boundary edge, we define τ_i^j as a negative value depending on the boundary conditions. Note that the neighbor relation need not be symmetric and is time-dependent.

To ensure that condition (ii) above is met and that the "clean up" will involve only degenerate quadrilaterals, we force a regular division of a triangle t_i whenever two of its neighbors have been divided once or one of its neighbors has been divided twice (cf. Figure 2.3). This guarantees that triangle sizes change by at most a factor of four between adjacent triangles (allowing for green triangles) and that at most one vertex can exist along the interior of any edge of an unrefined triangle.

Algorithm 2.1 is a high-level version of our refinement procedure.** If we view Ω itself as a "pseudo-triangle" t_0 that is the father of each of the triangles in $\mathcal{T}_0$, then the structure of the triangulation of Ω can be represented as a "triangle tree" in which the root is t_0, other vertices correspond to macro-triangles

---

* To simplify our notation, we sometimes let τ_i^j stand for the global triangle number of the neighbor of t_i across edge ε_i^j. However, this will be clear from context.

** In Algorithm 2.1, maxt is assumed to be an integer whose value bounds the maximum number of triangles of any type that may be created by REFINE. Although it is not explicitly indicated, DIVIDE and GREEN terminate in an error condition if too many triangles would be created.

or to triangles created during refinement, and edges lead downward from a triangle to its sons. All internal nodes of the tree have exactly four sons, except that t_0 has nt_0 sons, and fathers of green triangles have two sons. Leaves of the tree correspond to unrefined triangles. The level of a triangle t_i, denoted ℓ_i, is defined to be the distance from t_i to t_0 in the tree: that is, the length of a shortest path from t_i to t_0 in the tree.

For any given triangle t_i, certain standard information relating to its status on the triangle tree must be obtainable during the refinement process. This information is described below in the form of definitions of functions whose values are the standard information.

(i) knots:

knots(j,i) = k where $\nu_i^j = v_k$, $1 \leq j \leq 3$

(ii) neighbors:

$n(j,i) = \tau_i^j$, $1 \leq j \leq 3$*

(iii) father:

$f(i) = k$, where t_k is the father of t_i in the tree

(iv) son:

$$s(i) = \begin{cases} s_i & \text{if } t_i \text{ is normally refined} \\ 0 & \text{otherwise} \end{cases}$$

(v) level:

$\ell(i) = \ell_i$

---

* If t_i is a green triangle, then $n(j,i)$ is defined only if ε_i^j is a boundary edge.

(vi) macrofather/macrofather edge:

$mf(i) = k$ where t_k is a user specified triangle and $t_i \subseteq t_k$

$$mfe(j,i) = \begin{cases} m & \text{if } \varepsilon_i^j \text{ is a boundary edge and } \varepsilon_i^j \subseteq \varepsilon_{mf(i)}^m \\ 0 & \text{otherwise} \end{cases}$$

For a given vertex v_k, we must be able to generate the standard information described below:

(vii) vertex fathers:

$$vf(j,k) = \begin{cases} k & \text{if } k \leq nv_0 \\ m_j & \text{if } k > nv_0 \text{ and } v_k \text{ was obtained as the midpoint of the edge with endpoints } v_{m_j},\ j=1,2 \end{cases}$$

(viii) vertex type

$$ibc(k) \begin{cases} < 0 & \text{if } v_k \text{ is on } \partial\Omega \\ \geq 0 & \text{otherwise} \end{cases}$$

(ix) coordinates:

$vx(k)$ = x-coordinate of v_k
$vy(k)$ = y-coordinate of v_k

If (i)-(ix) above are known, one may carry out the standard computations required of a finite element program (e.g. determine the graph of the matrix, assemble the element stiffness matrices and right hand sides), as well as some of the more unusual computations in PLTMG (e.g. multi-level iteration, and procedure REFINE itself). Details will be provided elsewhere.

In early versions of PLTMG, knots, n, f, s, ℓ, mf, vf, and ibc were realized as integer arrays and vx and vy as real arrays, so that most of the standard information was simply looked up. Since the ratio of triangles on the tree to vertices is asymptotically

about 8/3, this meant that about 29 2/3 nv* integer and 2 nv real storage words were required in addition to the storage necessary for matrices, right hand sides and the solution.

In our current data structure this has been reduced to about 11 nv, all integer. The penalty associated with this reduction is that almost all the quantities must be computed as required. Our data structure is designed to allow the standard information to be computed in constant time (i.e., at worst, only a constant factor more costly than table look-up). In particular, we avoid extensive searching in the triangle tree, since a search from t_i to t_0 would run in time proportional to $\ell(i)$.

To conclude this section we discuss the way that Algorithm 2.1 fits into the multi-level solution scheme. Applying the algorithm to the initial triangulation causes a sequence of regular triangle subdivisions to occur, eventually leading to a fully refined grid. However, we can stop the process early by limiting the maximum allowable triangle level number (through DVTEST) and, after adding necessary green edges, obtain a partially refined triangulation suitable for use with the multi-level scheme. To obtain the necessary sequence of such triangulations, we simply make use of an increasing sequence of limits of the triangle level numbers. At each step we must logically remove any green edges in the current triangulation before invoking REFINE, but this is accomplished automatically because the structure of REFINE itself causes existing

---

* nv denotes the total number of vertices in the mesh.

green triangles to be ignored during regular refinement. The result is an efficient procedure that generates a sequence of triangulations of Ω which are nested (except for a few green triangles) and which satisfy the other constraints requisite to their use with the multi-level solution scheme (cf. Bank and Dupont [3]).

Algorithm 2.1

```
Procedure REFINE
    {If this is first call to REFINE Then [nt←nt_0; ng←maxt + 1];
     i←1;
     While (i ≤ nt) do
          [For j←1 to 3 do
               [If τ_i^j is undivided Then
                    [If τ_i^j has two divided neighbors Then DIVIDE (τ_i^j);
                     Else If ℓ_i > ℓ_{τ_i^j} + 1 Then DIVIDE (τ_i^j)]];
           If DVTEST (t_i) Then DIVIDE (t_i);
           i←i + 1];
     imax←nt;
     For i←1 to imax do
          [If t_i is undivided Then
               [If t_i has a divided neighbor Then GREEN (t_i)]]};
Procedure DIVIDE (t_i)
    {s_i←nt + 1;
     nt←nt + 4;
     Create t_{s_i+j}, 0 ≤ j ≤ 3, along with associated vertices};
Procedure GREEN (t_i)
    {s_i←ng - 2;
     ng←s_i;
     Create t_{s_i} and t_{s_i+1}};
Logical Function DVTEST (t_i)
    {...};
```

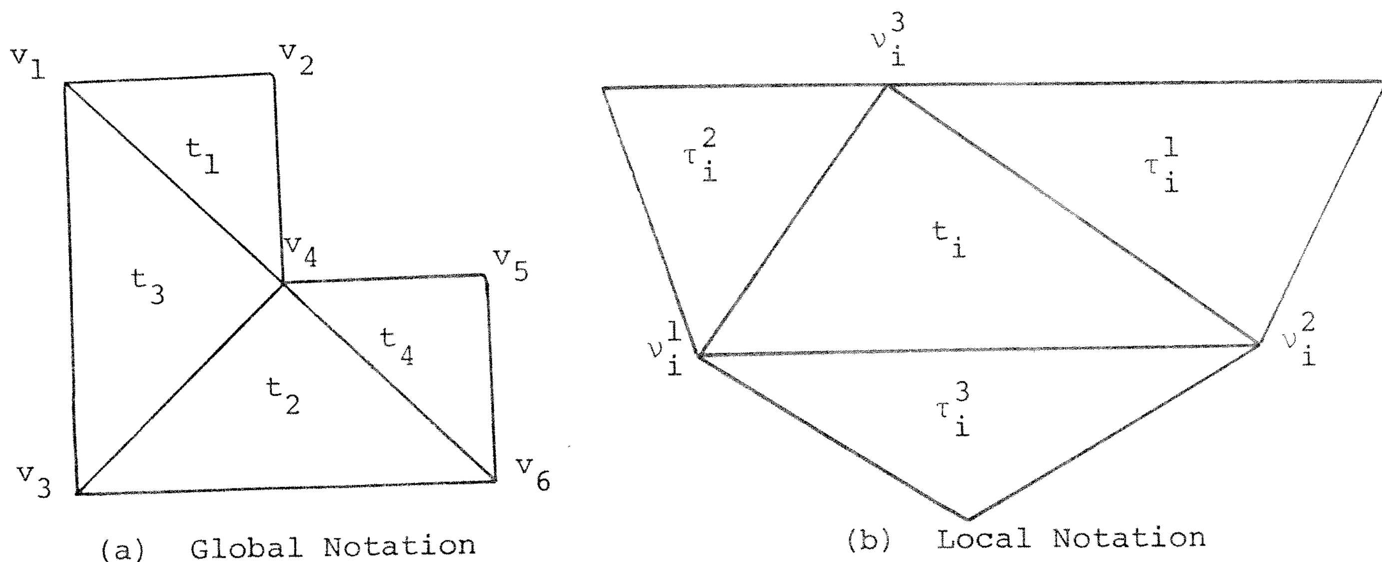
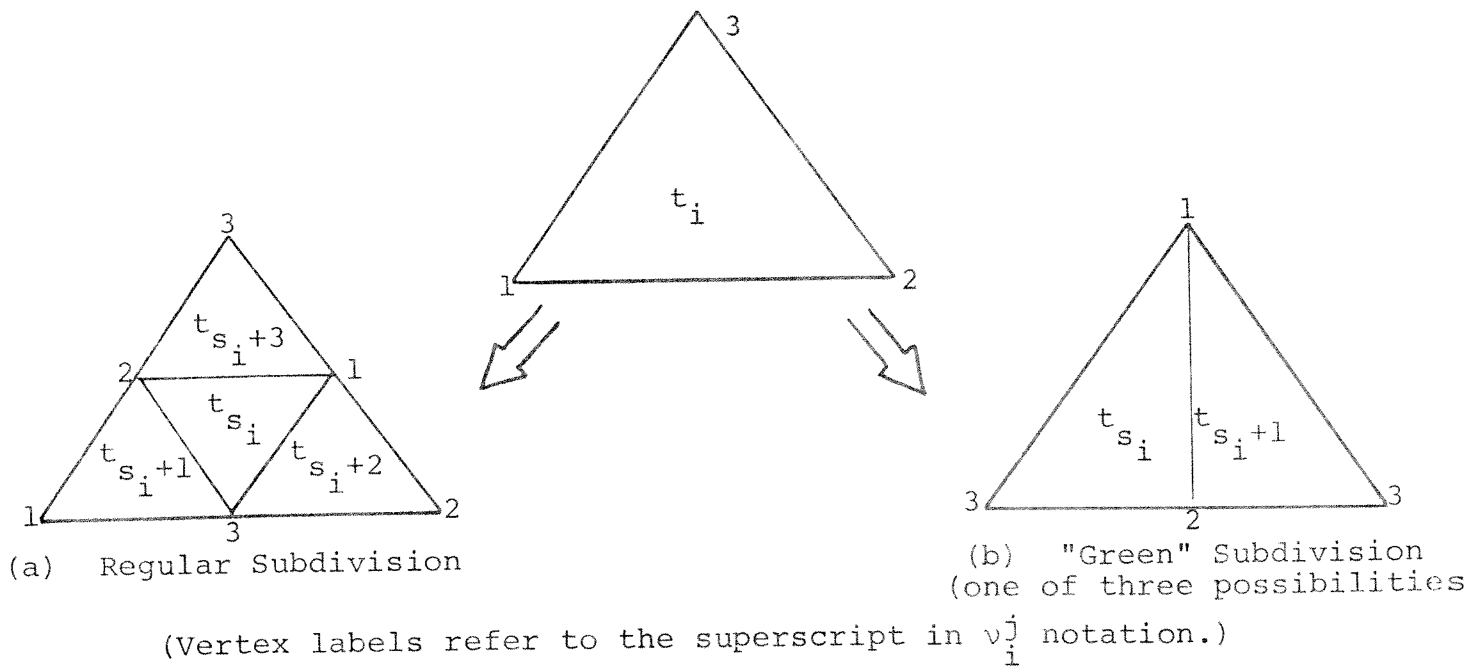


Figure 2.1: Triangle Notation



(Vertex labels refer to the superscript in ν_i^j notation.)

Figure 2.2: Types of Subdivision

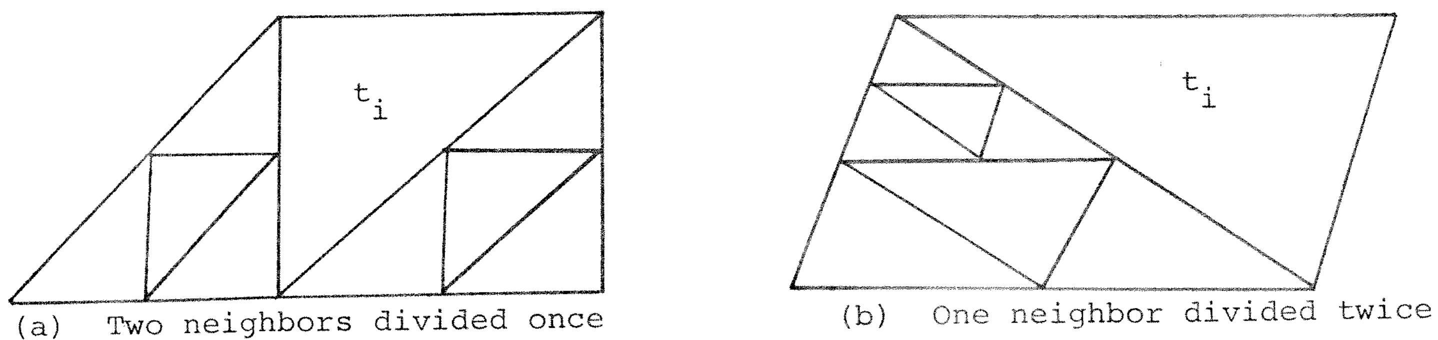


Figure 2.3: Situations requiring regular subdivisions of t_i

3. Definition of the Data Structure

The principal structure information describing the triangle tree in PLTMG is contained in two integer arrays: ITRI, of dimension 3 x maxt, and IVERT, of dimension 3 x maxv.* (Asymptotically, we expect nt ≃ 8/3 nv, and choosing maxt ≃ 8/3 maxv yields a total storage requirement of about 11 maxv). In this section we define the entries of these arrays.

The columns of ITRI are partitioned into five blocks, as illustrated in Fig. 3.1.

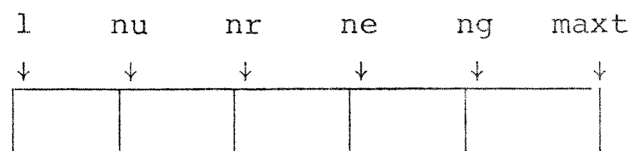


Figure 3.1: Partitioning of ITRI

The partitioning pointers nu, nr, ne, ng and maxt satisfy

$$nu > 4 \tag{3.1a}$$

$$nr - nu = nt_o \tag{3.1b}$$

$$nr = 0 \bmod 4 \tag{3.1c}$$

$$ne = 0 \bmod 2 \tag{3.1d}$$

$$ng = 0 \bmod 2 \tag{3.1e}$$

$$maxt = 1 \bmod 2 \tag{3.1f}$$

Columns 1 - (nu-1) are not used except as convenient storage for some constants (e.g. nu, nr, etc.). For convenience no triangles

---

* maxv is an integer variable whose value bounds the total number of vertices in the mesh.

on the tree are given numbers 1 - (nu-1). Columns nu - (nr-1) store information about user triangles with triangle t_{i+nu-1} corresponding to the ith user triangle. Columns nr - (ne-1) correspond to triangles obtained by regular subdivision, generated in groups of four during the refinement process, with column numbers in ITRI corresponding to triangle number. Using Fig. 2.3a and (3.1), note that i mod 4 can be used to determine the geometric relation of t_i to its father. If $i \bmod 4 = j \neq 0$ then t_i shares node $v^j_{f(i)}$ with $t_{f(i)}$.

Columns ng - maxt correspond to green triangles. Because of (3.1), the value of i mod 2 can be used to find the relation of t_i to its father. Columns ne - (ng-1) form the pool of space available to store information about newly created triangles. When new regular triangles are added, ne is increased by 4; when green triangles are added, ng is reduced by 2. When ne $\geq$ ng, no further triangles may be created. Initially nr = ne and ng = maxt + 1.

The information stored in ITRI for a triangle t_i differs depending on whether t_i is one of the user-specified macro triangles, one of a quartet of regular triangles, or one of a pair of green triangles. If t_i is a macro triangle, then the contents of the ith column of ITRI are as shown in Fig. 3.2.*

	i
1	
2	mf(i)
3	son(i)

Figure 3.2

* Blank fields are unused.

Due to our relabeling of user triangles, $mf(i) \neq i$. The son field contains the value s(i), unless t_i has been refined into a pair of green triangles, in which case it contains the number of the first triangle in the pair. Since, in general, there need be no additional structure to a user defined mesh, we augment ITRI with two small integer arrays ITNODE and ITEDGE, each of dimension $3 \times nt_0$. These arrays satisfy ITNODE(j,i) = knots(j,i) and ITEDGE(j,i) = n(j,i), respectively.

If t_{i+j}, $0 \leq j \leq 3$, form a quartet of regular triangles, then columns i - (i+3) of ITRI contain the information shown in Figure 3.3.

	i	i+1	i+2	i+3
1	f(i)	knots(1,i)	knots(2,i)	knots(3,i)
2	mf(i)/ℓ(i)	knots(1,f(i))	knots(2,f(i))	knots(3,f(i))
3	son(i)	son(i+1)	son(i+2)	son(i+3)

Figure 3.3

In the 12 available locations, we store the six relevant global vertex numbers, four son pointers (whose interpretation is the same as in the case of macro triangles), a father pointer, and either the macrofather or level. (During the refinement process, the level is stored; during element assembly, equation solution, and subsequent phases of computation, the macrofather is stored.)

If t_{i+j}, $0 \leq j \leq 1$, form a pair of green triangles, then columns i - (i+1) of ITRI contain the information shown in Figure 3.4.

	i	i+1
1	f(i)	knots(3,i)
2	knots(2,i)	knots(1,i)
3	s(f(i))	knots(3,i+1)

Figure 3.4

Five of the six entries are the four relevant global vertex numbers (column i+1 contains the vertices of $t_{f(i)}$ in permuted order) and f(i). The entry in location (3,i) requires some explanation. Because of the multi-level iterative method used in PLTMG, and the resulting necessity of having a sequence of triangulations, it is possible that in one triangulation, $t_{f(i)}$ has two green sons (t_i and t_{i+1}), while in a subsequent refinement $t_{f(i)}$ is refined regularly. (The triangulations will not be nested.) In this event the son field of $t_{f(i)}$ points to t_i, the first green son in the coarser triangulation, and the son field of the green triangle t_i points to $t_{s(f(i))}$, the first regular son of $t_{f(i)}$ in the finer triangulation.

We now describe the array IVERT. Column k of IVERT corresponds to vertex v_k in the grid. There are four different classes of vertices: user, green, regular, and boundary.* The first class contains the user-specified vertices, while the members of the other three classes are created by the regular refinement process. The entries of IVERT differ depending on the nature of v_k, as shown in Figure 3.5. If v_k is a user vertex, the first two entries of column k of

---

* User vertices on the boundary of Ω are classified as user vertices.

	user	green	normal	boundary
1		IVF1	IVF1	IVF1
2		F	IVF2	MFE
3	V	V	V	V

Figure 3.5

IVERT are unused; the third contains ibc(k). Additionally we require two short real arrays, VX and VY, of length nv_0, containing the x- and y- coordinates of the user vertices.

When a new vertex v_k is created in the interior of Ω, the situation is typically one of those shown in Figure 3.6. In both cases, the creation of v_k is required by the creation of the quartet of regular triangles t_{i+j}, $0 \leq j \leq 3$.

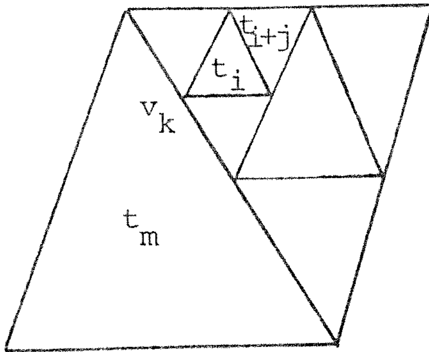


Figure 3.6a

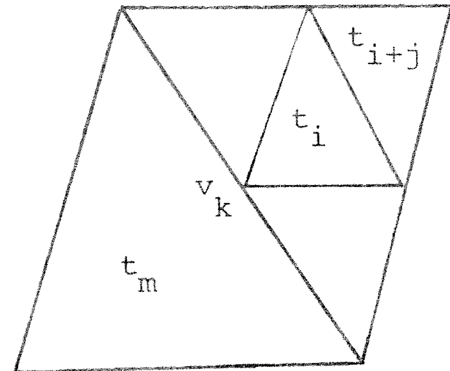


Figure 3.6b

In Figure 3.6a the creation of v_k will cause the eventual refinement of t_m, but in either case v_k is referred to as a green vertex. The IVF1 field is set to i+j where $1 \leq j \leq 3$ and i mod 4 = 0. Note that v_k is the midpoint of edge $\varepsilon^j_{f(i)}$ of triangle $t_{f(i)}$. The F field is set to -m, and the V field is set to ℓ, where v_k lies on edge ε^ℓ_m of triangle t_m. If the situation is as in Figure 3.6a, the F and V

fields must be updated when t_m is refined, but v_k will still be a green vertex at that point. Eventually, if the refinement process is continued, v_k may become a vertex of the center triangle in a second quartet of regular triangles (e.g. if triangle t_m in Figure 3.6b is refined regularly). Then v_k becomes a normal vertex, and the IVF2 field is set in the same way as the IVF1 field, but for the second quartet.

A vertex v_k created on a boundary edge of $\partial\Omega$ is called a boundary vertex. The IVF1 field is set as described for green vertices, the V field contains a value indicating the type of boundary conditions that apply to v_k, and MFE is set to j where v_k is on edge ε_m^j of macro triangle t_m.

The actual refinement process consists of filling in the entries of ITRI and IVERT. When a triangle is refined regularly, all of the information for ITRI, with the exception of the global vertex numbers of the center triangle, is trivially known. For the center triangle, one must either create new green or boundary vertices or change current green vertices into normal ones. To do this one must have the ability to compute knots(j,i), n(j,i) (for $1 \leq j \leq 3$), f(i), s(i), and ℓ(i) for any triangle t_i currently on the tree. When a vertex is created or changes status, all the necessary information for IVERT is on hand, having been generated in the process of determining whether or not the vertex existed as a green vertex.

When the regular refinement process stops, each remaining green vertex requires the creation of two green triangles. Since the F field of a green vertex points to the father of the green triangles, all of the necessary information for ITRI is immediately available.

4. Algorithms for Computing the Standard Information

In this section we describe the algorithms which we use to compute the standard information about the grid.

4.1 knots(j,i)

If t_i is a macro triangle then knots(j,i) is found by table look-up in the array ITNODE. If t_i is a regular triangle then the value $k \equiv i \mod 4$ can be used to determine the position of t_i relative to $t_{f(i)}$. Once this is known, the appropriate subset of global vertex numbers stored in columns i-k to i-k+3 of ITRI can be determined.* If t_i is a green triangle one computes the value $k \equiv i \mod 2$, and the appropriate subset of global vertex numbers stored in columns i-k to i-k+1 of ITRI can be determined.*

4.2 n(j,i) and mfe(j,i)

If t_i is a macro triangle, n(j,i) is found by table look-up in the array ITEDGE. If t_i is a regular triangle and $i \mod 4 = 0$, then $n(j,i) = i+j$, as is evident in Figure 2.3. If t_i is a regular triangle and $k \equiv i \mod 4 \neq 0$, then $n(k,i) = i-k$ (i.e. the center triangle). The other two values of n(j,i) are found by considering the two vertices $(\nu_i^{j_1}, \nu_i^{j_2})$ which t_i shares with t_{i-k}. If a vertex $\nu_i^{j_m}$ is a green vertex, then the corresponding F field in IVERT points to $n(j_{3-m}, i)$. If a vertex $\nu_i^{j_m}$ is a regular vertex, then either the IVF1 or IVF2 field in IVERT points back to one of the quartet of triangles containing

---

* It is easy to see that t_{i-k} is the first triangle of the quartet or pair containing t_i.

t_i; the other field points to the second quartet of triangles whose center triangle contains the given vertex; $n(j_{3-m},i)$ can then immediately be determined. If a vertex $\nu_i^{j_m}$ is a boundary vertex, then $n(j_{3-m},i)$ is set to the appropriate boundary information.

If t_i is a green triangle, then we need only compute n if t_i has a boundary edge. From Figure 2.3 it is clear that the only possible boundary edge of a green triangle is edge ε_i^2 (green knots cannot occur on the boundary), and this edge is shared with $t_{f(i)}$. If ε_i^2 is, indeed, a boundary edge, then n(2,i) is set to the appropriate boundary information.

In the course of computing n(j,i), it is determined whether or not ε_i^j is on the boundary. With this information the computation of mfe(j,i) is trivial. If t_i is a macro triangle mfe(j,i) = j. If t_i is a regular triangle, then mfe(j,i) is found in the MFE field of the appropriate boundary vertex. If t_i is a green triangle, then mfe(2,i) can be found by examining $t_{f(i)}$. In any event, the algorithms for computing n and mfe are so closely related, a single routine in PLTMG is used to compute both of them.

4.3 f(i)

If t_i is a macro triangle, then f(i) = 0. If t_i is a regular triangle and $k \equiv i \bmod 4$, then f(i) = ITRI(1,i-k).

4.4 s(i)

If t_i is a macro or regular triangle without a pair of green sons, then s(i) = ITRI(3,i). On the other hand, if t_i does have green sons, then s(i) is found in the son field for the pair of green triangles

4.5 $\ell(i)/mf(i)$

The refinement procedure in Section 2 requires $\ell(i)$, but not $mf(i)$. All of the procedures in PLTMG applied subsequent to refinement do not require $\ell(i)$, but several require $mf(i)$. This is the reason that $\ell(i)$ and $mf(i)$ can share the same entry in ITRI. In this subsection, we discuss the conversion of that entry from level data to macro father data.

If t_i is a macro triangle, then $\ell(i) = 1$ and $mf(i) = i-nu+1$ (cf. Fig. 3.1). If t_i is a regular triangle and $k \equiv i \bmod 4$, then ITRI(2,i-k) contains either $mf(i)$ or $\ell(i)$. If t_i is a green triangle, then $\ell(i) = \ell(f(i))+1$ and $mf(i) = mf(f(i))$.

To convert ITRI from storage of level data to storage of macro father data, or vice-versa, is quite simple. $\ell(i)$ and $mf(i)$ are both trivially known for macro triangles. Then it is simple to compute $\ell(i) = \ell(f(i))+1$ or $mf(i) = mf(f(i))$ for the center triangle of each quartet of regular triangles, in the order in which the quartets were created. (The indices of the appropriate columns of ITRI run from nr to ne in steps of four; cf. Fig. 3.1.)

4.6 $vf(j,k)$

If v_k is a user vertex then $vf(j,k) = k$, $1 \leq j \leq 2$. If v_k is not a user vertex then the IVF1 field of IVERT for v_k points to a quartet of triangles, among whose vertices are v_k and the two vertex fathers of v_k. By examining the value (mod 4) of this pointer, it is possible to deduce the vertex fathers. (The order of the vertex fathers is unimportant since they are always used in pairs.)

4.7 ibc(k)

ibc(k) is always found as IVERT(3,k). The precise values stored for ibc(k) are irrelevant here, but they are described in the users' guide for PLTMG [5].

4.8 vx(k), vy(k)

In PLTMG the only times that the x- and y- coordinates of a given vertex are actually needed are when it is necessary to evaluate the coefficient functions of the partial differential equation at a particular value of (x,y) (e.g., in assembling the stiffness matrix and right hand side). In such situations we take two temporary storage arrays of length nv and compute vx(k) and vy(k) for each value of k. For $1 \leq k \leq nv_0$, these values are looked up in the arrays VX and VY. For $nv_0 < k \leq nv$ (increasing order), we compute vf(j,k), $1 \leq j \leq 2$. Since $vf(j,k) < k$, and $k > nv_0$, the x- and y- coordinates of the vertex fathers have previously been computed. Hence it is easy to compute vx(k) and vy(k) since v_k is the midpoint of the edge between its vertex fathers.

The time spent computing all of the x- and y- coordinates before assembling a stiffness matrix is a very small fraction of the time spent in such a routine. By recomputing vx(k) and vy(k) whenever they are needed, we save storage since the temporary storage used can be shared with several other routines in which such storage is also required.

4.9 Running Times

The algorithms for knots, n, f, s, ℓ, mf, vf, and ibc all run in constant time. The procedures for converting ITRI from level data to macro father data and back run in time proportional to the number of triangles; but they are used only as post- or pre- processing steps to Algorithm 2.1, which itself runs in time proportional to the number of triangles (with a much larger constant). The algorithms used to compute vx(k) and vy(k) run in time proportional to nv, but they are always used as pre-processing steps to procedures that run in time proportional to nv with much larger constants. The net result is that at the cost of only a little extra time (over that required by table look-up techniques), it is possible to save a lot of storage, thereby allowing the solution of larger problems in core.*

---

* To balance the slight increase in time noted here, we remark that an efficient depth-first search procedure for the triangle tree can be carried out using s(i) and f(i).

5. Rectangular Elements

Most, but not all, of the ideas presented in sections 2-4 generalize immediately to the case of C^0-piecewise bilinear rectangular elements. We believe that the result is somewhat cleaner and simpler than similar data structures proposed elsewhere (e.g. [6]). For a given rectangle t_i, we can define local labels for the sides and vertices as indicated in Figure 5.1.

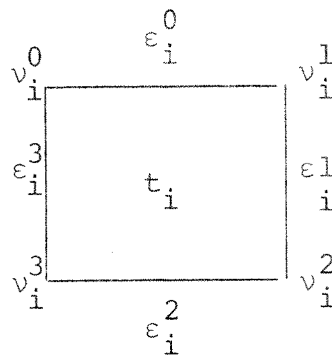


Figure 5.1

If Ω is composed of the union of rectangular elements, we can give meaning to the terms "top", "bottom", "leftside" and "rightside" and thus label all elements to be consistant with each other as well as consistant internally. This simplifies greatly the neighbor relation in comparison with the case of triangular elements. For example, rectangle τ_i^0 will have edge $\varepsilon^2_{\tau_i^0}$ in common with ε_i^0.

When a rectangle t_i is refined, its four sons t_{s_i+j}, $0 \le j \le 3$, are labeled consistantly as shown in Figure 5.2.

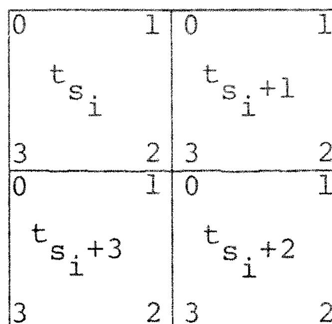


Figure 5.2*

Note that t_{s_i+j} shares vertex $\nu^j_{s_i+j}$ with its father t_i. The refinement of a rectangle always requires the creation of at least one new vertex (the center), although the four edge midpoints may have been created by the refinement of the neighbors of t_i. We will require the center vertex to have the highest global vertex number of any of the nine vertices shared by the quartet of sons of t_i.

There is no obvious analogue of green triangles in the case of rectangles. We cannot clean up exposed green knots by just adding edges since that would only lead to new green knots. Rather, we must require green knots to satisfy interpolation conditions which will guarantee the conformity of the finite element subspace. For piecewise bilinear elements, a green knot will not introduce a new degree of freedom; instead we require the solution at a green knot to be the average of the solution at its two vertex fathers, i.e.,

$$u(v_g) = 1/2(u(v_{f_1}) + u(v_{f_2}))$$

(cf. Figure 5.3).

---

* Here we use the vertex label k as shorthand for ν^k_p in rectangle t_p, $s_i \leq p \leq s_i+3$.

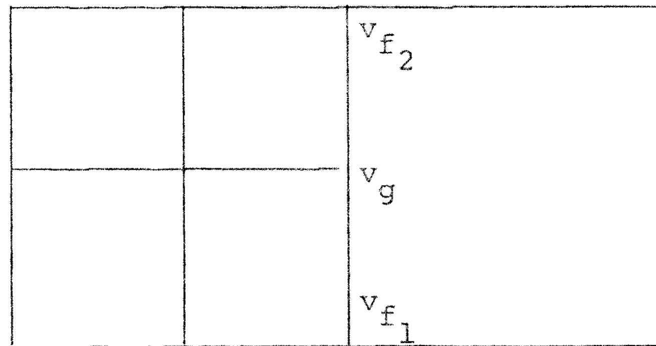


Figure 5.3

This insures C^0 continuity of the finite element subspace. The net result is that the problem of dealing with green knots is transferred from the refinement procedure to the matrix and right hand side assembly procedures.*

In analogy with Figure 2.3b, we require the refinement of t_i whenever any son of a neighbor of t_i has been refined, as illustrated in Figure 5.4.

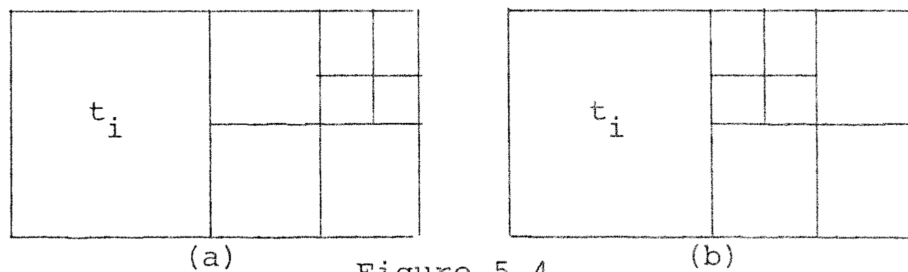


Figure 5.4

This rule insures that if v_k is a green knot, then neither of its two vertex fathers is also green. This effectively limits the search which is required to determine the (at most five) nonzero basis functions in a given element. (They will be associated with the four vertices or their vertex fathers.) Since the center vertex and the

---

* This approach could also have been taken for green knots in the case of triangular elements. However, in that case, the creation of green triangles is more satisfactory since it maintains the number of degrees of freedom without any increase in the number of knots.

vertex a given rectangle shares with its father cannot be green, a given rectangle can have at most two green vertices, at opposing corners.

The analogue of procedure refine could now be formulated without the analogue of the refinement rule depicted in Figure 2.3a. However, it is advantageous to refine t_i whenever at least three of its neighbors have been refined (cf. Figure 5.5).

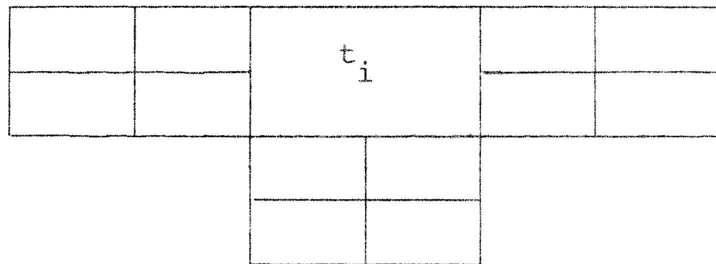


Figure 5.5

If three neighbors have been refined, adding only two new knots adds four degrees of freedom; if all four neighbors have been refined adding one new knot adds five degrees of freedom.

The analogue of Algorithm 2.1 can now be formulated as in Section 2. The resulting algorithm is actually somewhat simpler than Algorithm 2.1 since the sections dealing with green triangles will be deleted. (We do not present the algorithm here.)

In terms of data structures, we will require a 4 x maxt integer array IRCT, and a 2 x maxv array IKNOT.* IRCT will be partitioned in a similar fashion to ITRI, as shown in Figure 5.6.

---

* maxt and maxv have the same role here as for the arrays ITRI and IVERT.

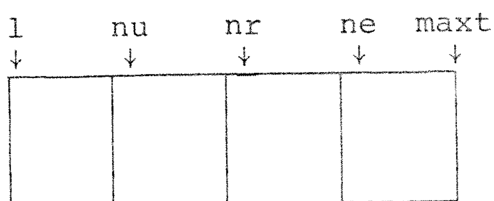


Figure 5.6: Partitioning of IRCT

As before, we require certain properties of the partitioning parameters:

$$nu > 4 \tag{5.1a}$$

$$nr - nu = nt_0 \tag{5.1b}$$

$$nr = 0 \bmod 4 \tag{5.1c}$$

$$ne = 0 \bmod 4 \tag{5.1d}$$

$$maxt = 3 \bmod 4 \tag{5.1e}$$

For a user-specified macro rectangle, with internal rectangle number i, $nu \leq i \leq nr-1$, we would store s(i) and mf(i) in two of the four locations of column i of IRCT (cf. Figure 5.7). Once again, $mf(i) \neq i$, in general.

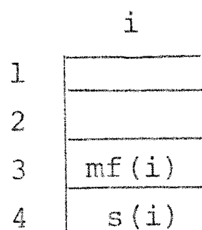
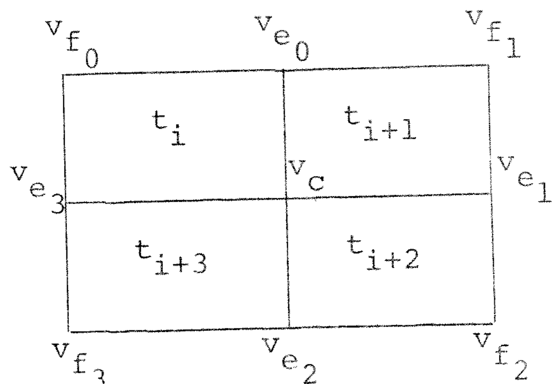


Figure 5.7

We would also require two integer arrays of size 4 x nt_0 to store knots(j,i) and n(j,i), $0 \leq j \leq 3$, $nu \leq i \leq nr-1$.

For a quartet of four rectangles obtained through refinement, the 16 available locations in IRCT will contain the nine global

vertex numbers, f(i), mf(i), ℓ(i) and s(i+j), $0 \le j \le 3$, as depicted in Figure 5.8. The mf and ℓ fields need not be shared as in the case of triangles.



	i	i+1	i+2	i+3
1	f(i)	f_o	e_o	f_1
2	ℓ(i)	e_3	c	e_1
3	mf(i)	f_3	e_2	f_2
4	s(i)	s(i+1)	s(i+2)	s(i+3)

Figure 5.8

As in the case of triangles, i mod 4 = 0.

In the IKNOT array there are five classes of vertices - user, green, normal, boundary, and center. The fields for the various types of vertices are illustrated in Figure 5.9.

	user	green	normal	boundary	center
1	0	-IVF1	IVF1	IVF1	IVF1
2	ibc(i)	F	IVF2	ibc(i)	0

Figure 5.9

All the fields for green, regular, and boundary vertices are filled in just as for the array IVERT for triangular meshes. A boundary vertex can be distinguished because $ibc(i) < 0$ for boundary vertices, while IVF2 and F are larger than four by (5.1). If v_k is a center vertex its IVF1 field points to t_i, the upper left rectangle of the quartet, so that IVF1 = 0 mod 4 for a center knot. To distinguish a center knot, IKNOT(2,k) = 0, since 0 cannot be the second entry for any other vertex class.

The algorithms for computing knots(j,i) and n(j,i), $0 \leq j \leq 3$, f(i), s(i), ℓ(i), mf(i) are the obvious generalizations, or, in some cases, simplifications, of the corresponding algorithms for triangular elements. The computation of mfe(j,i) is trivial since all rectangles are consistantly labeled. The computation of ibc(i) is similar to the triangle case.

vf(j,k) is defined and generated as in the triangle case for all vertices except center vertices. In some sense, such a vertex really has four vertex fathers, but in almost all situations the two vertex fathers of a center vertex of rectangle t_i can be taken as either pair of edge midpoints (top and bottom or left and right) in t_i. Since we have required the center vertex to have the largest of the nine global vertex numbers, the center vertex will have a larger vertex number than either of its vertex fathers. Thus all the usual applications of vf(j,k) (e.g. computing vx(k) and vy(k)) will work exactly as in the case of triangles.

Finally, it is worth remarking that the ratio of rectangles on the rectangle tree to vertices is about 4/3 as opposed to 8/3 for the case of triangles. Thus the storage required for IRCT and IKNOT is about 7 1/3 nv as opposed to 11 nv for ITRI and IVERT. This demonstrates that rectangles are asymptotically more storage efficient than triangles if the measure of efficiency is words of storage per knot. Note, however, that for finite element work, the number of degrees of freedom is a more important parameter than the number of knots, and rectangles may not be more storage efficient if the measure of efficiency is words of storage per degree of freedom (since green knots reduce the number of degrees of freedom).

Acknowledgement

We are pleased to acknowledge the contribution of Alan Weiser to our understanding of the application of our ideas to rectangular elements.

References

[1] I. Babuska and W. C. Rheinboldt. Error estimates for adaptive finite element computations. SIAM J. Numer. Anal. 15 (1978), pp. 736-754.

[2] R. E. Bank. A comparison of two multi-level iterative methods for nonsymmetric and indefinite finite element equations. SIAM J. Numer. Anal., to appear.

[3] R. E. Bank and T. Dupont. An optimal order process for solving finite element equations. Manuscript, Department of Mathematics, University of Texas at Austin, August 1979.

[4] R. E. Bank and A. H. Sherman. Algorithmic aspects of the multi-level solution of finite element equations. Sparse Matrix Proceedings 1978, I. S. Duff and G. W. Stewart, editors, SIAM Press, Philadelphia, 1979, pp. 62-89.

[5] R. E. Bank and A. H. Sherman. PLTMG user's guide. Report 152, Center for Numerical Analysis, University of Texas at Austin, September 1979.

[6] W. C. Rheinboldt and C. K. Mesztenyi. On a data structure for adaptive finite element mesh refinements. Trans. Math. Soft. 6 (1980), pp. 166-187.

[7] R. B. Simpson. A survey of two-dimensional finite element mesh generation. Manuscript, Department of Computer Science, University of Waterloo, December 1979.